Noemí Caballero Sánchez

Implementación en lenguaje Perl de algoritmos de recuento de *k*-meros y su aplicación al ensamblaje de novo de genomas

Trabajo tutorizado por: Dr. Héctor Candela Antón
Departamento de Biología Aplicada, Área de Genética

Grado en Biotecnología
Facultad de Ciencias Experimentales
Universidad Miguel Hernández de Elche
Curso 2016-2017

ÍNDICE DE MATERIAS

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. Resumen

En este trabajo, hemos perfeccionado un programa para el ensamblaje *de novo* de secuencias nucleotídicas basado en grafos bidirigidos de De Bruijn. Entre las mejoras que hemos introducido, destaca la posibilidad de realizar el recuento de *k*-meros mediante tres algoritmos distintos. El recuento de *k*-meros es una etapa crítica en cualquier programa de ensamblaje *de novo* basado en grafos. Además, los algoritmos seleccionados permiten optimizar la cantidad de memoria utilizada ya que se emplean distintas estrategias para descartar los *k*-meros que, presumiblemente, contienen errores de secuenciación. Hemos utilizado algoritmos que utilizan filtros de Bloom o crean particiones de los datos para hacer el recuento de los *k*-meros. Para comprobar la eficacia de los algoritmos implementados, hemos realizado ensamblajes *de novo* de los genomas de *Escherichia coli* y *Arabidopsis thaliana.*

Palabras clave: ensamblaje de genomas, filtros de Bloom, algoritmos de recuento de *k*-meros, grafos de De Bruijn bidirigidos.

In this work, we have improved a program for *de novo* assembly of nucleotide sequences based on the use of bidirected De Bruijn graphs. Among the new developments, we highlight the possibility of counting *k*-mers through three different algorithms. Counting *k*-mers is a critical step for all *de novo* assembly programs that use graphs to represent the sequences. In addition to this, the selected algorithms allow one to optimize the amount of memory required, as they use different strategies to discard *k*-mers derived from sequences containing sequencing errors. We have used algorithms that use Bloom filters or that partition the data in order to count the *k*-mers. To test the efficiency of the implemented algorithms, we have carried out de novo assemblies of the *Escherichia coli* and *Arabidopsis thaliana* genomes.

Keywords: genome assembly, Bloom filters, *k*-mer counting algorithms, bidirected de Bruijn graphs.

2. Introducción

2.1. Retos de la bioinformática

Desde la presentación de la secuencia del genoma humano en 2001, que se determinó mediante el método de Sanger (Craig *et al.,* 2001), y con el desarrollo de nuevas tecnologías de secuenciación masivamente paralela (*massively parallel sequencing technologies*), la bioinformática se ha visto obligada a innovar para manejar y procesar cantidades crecientes de datos (*big data*). Para conseguir este objetivo se ha optimizado el uso diversas estructuras de datos que permiten un mejor almacenaje y procesamiento de la información.

Entre los nuevos métodos de secuenciación destacan los basados en pirosecuenciación (Ronaghi *et al.,* 1998), ligación (SOLiD; Dressman *et al.,* 2003), síntesis con terminadores reversibles (Illumina; Fedurco *et al.*, 2006; Turcatti *et al.,* 2008) o el uso de semiconductores (Ion Torrent; Rothberg *et al.,* 2011). Estas nuevas tecnologías de secuenciación se caracterizan por su funcionamiento en paralelo, un mayor rendimiento y un menor coste, pero las secuencias obtenidas (denominadas lecturas) suelen ser de menor longitud que las del método de Sanger, lo que dificulta el ensamblaje del genoma (Miller *et al.,* 2010). Así mismo, conlleva la necesidad de una mayor cobertura (número medio de veces que se ha secuenciado cada nucleótido) para evitar errores de secuenciación (Edgar *et al.,* 2015; Alkan *et al.,* 2011). Por ello, se han creado nuevos programas que solventen la complejidad de estas tareas aunque algunos conceptos como, solapamientos o cóntigos, se habían definido a finales de la década de 1970 (Staden, 1979).

2.2. Estructuras de datos

Una estructura de datos es una manera de organizar los datos, de modo que un ordenador pueda recuperar, procesar y almacenar la información de manera eficiente (Joyanes-Aguilar, 2008). Las estructuras de datos pueden clasificarse según su linealidad. Los vectores son estructuras de datos lineales, en las que los elementos se almacenan secuencialmente. Los grafos, los árboles y las tablas *hash* son ejemplos de estructuras no lineales. En las siguientes secciones se describen las principales estructuras de datos utilizadas por los algoritmos implementados en este trabajo.

2.2.1. Vectores

Un vector es una estructura de datos utilizada habitualmente para almacenar de elementos del mismo tipo. Estos elementos se encuentran ordenados dentro de casillas, de tal forma que a cada casilla le corresponde un elemento. Así mismo, cuando se utiliza un vector hay que indicar el número de casillas total del mismo, por lo que el espacio del vector es finito. El uso de vectores facilita el acceso a los datos si se conoce la posición o índice del

elemento, pudiendo añadir, reordenar y eliminar elementos de un vector. (Joyanes-Aguilar, 2008).

En este trabajo se ha implementado el algoritmo `turtle`, explicado en el apartado 3.4.2.1, que usa un método llamado un método llamado *Sorted And Compressed array* (SAC). Éste permite reordenar elementos dentro de un vector, que en el caso de turtle se ordenan alfabéticamente, y posteriormente hay un paso de compresión cuando dos elementos poseen la misma secuencia alfanumérica. Esto permite liberar espacio y que se puedan seguir añadiendo elementos al vector. Este procedimiento se llevará a cabo hasta que finalicen los datos de entrada o bien hasta que se complete todas las casillas del vector (Roy *et al.,* 2014).

2.2.2. Tabla *hash*

Una tabla *hash* es una estructura de datos en la que se almacenan parejas formadas por claves (*keys*) y valores. Las claves pueden ser un texto, un archivo, una contraseña, etc. A dichas claves se les aplica una serie de funciones *hash*, obteniéndose una salida alfanumérica. Las funciones *hash* son computables mediante un algoritmo y son irreversibles, por lo que, a partir de una serie de claves se genera una lista de datos de salida que solo puede volverse a crear con esos mismos datos (Sedgewick *et al.,* 2011).

Las tablas *hash* pueden usarse para determinar si dos objetos son iguales o para encontrar elementos en una base de datos. Una ventaja de las tablas *hash* es que permiten acceder a la información en tiempo constante, que no depende del tamaño de la tabla ni del número de elementos almacenados en la misma (Sedgewick *et al.,* 2011). Esta ventaja no está presente en los vectores, por lo que, a la hora de manejar los datos, una tabla *hash* va a conllevar un menor tiempo de ejecución.

2.2.3. Filtros de Bloom

Un filtro de Bloom es una estructura de datos probabilística diseñada por Burton Howard Bloom en 1970 (Bloom, 1970). Los filtros de Bloom constan de un mapa de *bits* y varias funciones *hash*. Los *bits* son dígitos binarios que definen la unidad mínima de almacenamiento en memoria.

Un filtro de Bloom vacío es una matriz de *bits*, todos establecidos en 0. También debe haber un cierto número de funciones *hash* definidas que activen ciertos *bits* de la matriz (dándoles un valor de 1), creando una distribución aleatoria uniforme. De esta manera, si un nuevo elemento se añade al filtro de Bloom y se activan las mismas posiciones de la matriz (ya activadas por la adición de un elemento incorporado anteriormente), se puede afirmar que, probablemente, el elemento aparece dos o más veces.

Los filtros de Bloom no pueden contener falsos negativos, pero sí falsos positivos debido a las colisiones en las funciones *hash* (Chu *et al.,* 2014). Estas colisiones ocurren cuando dos elementos distintos dan como resultado la activación de los mismos *bits* y, por tanto, son reconocidos como elementos idénticos. En la Figura 1 se muestra un ejemplo en el que se representan cuatro datos de entrada diferentes entre sí (*a, b, c* y *d*) y tres funciones *hash* (*h*). *a* y *b* son datos insertados en el filtro de Bloom, mientras que *c* y *d* no lo han sido. La inserción de *a* y *b* al filtro implica la activación de ciertos *bits*. Cuando se incorpora *c*, al tratarse de un nuevo elemento, no activa los mismos *bits* que *a* o *b*. En cambio, a pesar de que *d* también es un elemento distinto de *a*, *b* y *c*, activa las mismas posiciones *b*, de modo que el programa detectaría que *b* aparece dos veces y generándose así un falso positivo. En otras palabras, el uso de distintas funciones *hash* ha provocado que la combinación de las mismas activen un dato que no debería ser activado.

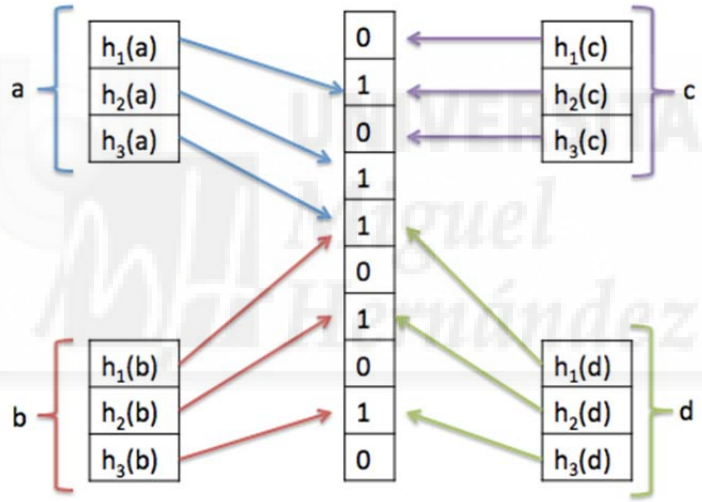

Figura 1. Imagen modificada a partir del artículo (Melsted *et al.*, 2011). Ejemplo de filtro de Bloom en el que se exponen tres casos y la explicación de su activación en el filtro. Donde *a,b,c* y *d* son datos de entrada y *h* son funciones *hash.*

2.2.4. Count-Min Sketch

Count-Min Sketch (Cormode *et al.,* 2005) es una estructura de datos probabilística parecida a los filtros de Bloom, aunque en este caso se define como una estructura de datos compacta de tipo vector. Dicho vector es una matriz cuyo espacio está determinado por la cantidad de datos al inicio, es decir, se rellena la matriz hasta completarla con todos los datos que se quieren analizar, denominados borrador o *sketch.* A partir de la matriz se generan vectores definidos por funciones *hash*, por lo que aumentar el rango de las funciones *hash* aumenta la precisión, pero al aumentar el número de posibles colisiones se generan más falsos positivos. Debido a que Count-Min Sketch utiliza vectores como

estructura de datos se mantiene la linealidad de los elementos, por lo que éstos pueden ser modificados o incluso se pueden combinar vectores si es necesario.

2.2.5. Árboles

Un árbol es una estructura de datos que se compone de un conjunto de vértices ordenados jerárquicamente. De esta forma, existe un vértice raíz y una colección de subárboles, o vértices hijos, derivados de la raíz y conectados por ramas. Se considera una estructura no lineal dado que un mismo vértice puede estar enlazado con uno o más vértices (Joyanes-Aguilar, 2008).

2.2.6. Grafos

Un grafo es una estructura de datos que son expresados como un conjunto de vértices unidos por aristas que permite estudiar la relación entre vértices (Joyanes-Aguilar, 2008). Una vez representado el grafo, se pueden realizar dos tipos de recorridos distintos: un recorrido hamiltoniano es aquel que recorre todos los vértices de un grafo sin pasar dos veces por el mismo vértice, mientras que un camino euleriano es aquel que recorre todas las aristas una única vez (Pevzner *et al.,* 2001). Así mismo, los grafos pueden ser “no dirigidos” lo que implica que las aristas son únicamente líneas que unen los vértices, o pueden ser dirigidos, en cuyo caso, lo que unen los vértices son flechas que aportan direccionalidad (Medvedev *et al.,* 2007; Xu *et al.,* 2005).

2.3. Técnicas para la secuenciación de genomas

La secuenciación de genomas puede resolverse por dos formas diferentes: resecuenciación y ensamblaje *de novo.* El ensamblaje de genomas se define como la reconstrucción de secuencias largas a partir de secuencias de lectura más cortas (Ekblom *et al.,* 2014).

La diferencia principal entre la resecuenciación y el ensamblaje *de novo* es que la primera se basa en comparar las lecturas obtenidas con un genoma modelo, mientras que en el ensamblaje *de novo* se reconstruye la secuencia a partir de las lecturas sin ningún dato externo. Esto conlleva que en el ensamblaje *de novo* no se descarten lecturas que están presentes en la muestra y que no se alinean con el genoma modelo, pero la secuenciación es más fragmentada y se obtiene una mayor cantidad de cóntigos en comparación con el sistema de resecuenciación.

2.3.1. Técnicas para la resecuenciación

La resecuenciación requiere un genoma de referencia, ya sea de la misma especie o de alguna muy cercana. Consiste en alinear lecturas con el genoma de referencia, lo que permite detectar las diferencias entre ambos. Permite el estudio de la variación genética entre individuos al mismo tiempo que incrementa el número de genomas secuenciados de

una misma especie en bases de datos. Es un método rápido que no requiere una cobertura muy alta de las lecturas.

Para realizar el alineamiento se hace uso de distintas estructuras de datos y técnicas para el manejo de las mismas, como los árboles de sufijos o la transformación de Burrows-Wheeler. En el caso de la resecuenciación, los principales programas como SOAP2 (Li *et al*., 2009), bowtie2 (Langmead *et al*., 2009) o BWA (Li *et al*., 2009) usan ambas técnicas para realizar el alineamiento de las secuencias.

2.3.2. Técnicas para el ensamblaje *de novo*

El ensamblaje *de novo* consiste en la reconstrucción de la secuencia sin un genoma de referencia con el que comparar, por lo que la memoria y el tiempo de ejecución serán mayores. Existen tres técnicas para la resolución del ensamblaje *de novo* (Nagarajan *et al.,* 2013; El-Metwally *et al.,* 2013): algoritmos voraces (*greedy*), solapamiento-disposición-consenso (*Overlap-layout-consensus*) y los grafos de De Bruijn (será esta última técnica la que aplicaremos para la representación del genoma). La elección de la técnica dependerá en gran medida del tamaño de las lecturas.

2.3.2.1. Algoritmos voraces

Los algoritmos voraces son aquellos algoritmos que escogen la opción más correcta para resolver un problema en casa paso. Aunque la solución final puede no ser la óptima, se supone que las elecciones en cada etapa conllevarán a la solución global correcta. Referido al ensamblaje *de novo*, en los algoritmos voraces se ensamblan las lecturas y se localizan las que alinearían mejor. Se realizan elecciones locales sobre los alineamientos teniendo en cuenta no alterar al conjunto ya construido. No obstante, el principal problema de estos algoritmos es que las elecciones realizadas son locales y no toman en cuenta la relación global entre las lecturas.

Muchos de los primeros ensambladores, como PHRAP (De la Baside *et al.,* 2007) y TIGR Assembler95 (Zhang *et al.,* 2000), se basaron en estos algoritmos. Sin embargo, en la actualidad no son ampliamente utilizados debido al problema comentado anteriormente sobre las elecciones locales.

2.3.2.2. Estrategia solapamiento-disposición-consenso

Es una técnica que se utiliza para el ensamblaje *de novo* y, consta de tres fases. Una primera fase denominada solapamiento (*overlap*) en la cual se identifican todos los pares de lecturas que solapan y se organiza esta información en un grafo. Los vértices del grafo corresponden a las lecturas y las aristas a los prefijos o sufijos resultado del solapamiento entre lecturas. Durante la segunda fase, conocida como disposición (*layout*), lo que se intenta es encontrar el recorrido hamiltoniano más corto. La estructura del grafo permite el desarrollo de complejos algoritmos de ensamblaje que tienen en cuenta la relación global

entre las lecturas, De esta manera, en la tercera fase o consenso (*consensus*), se definen caminos que se corresponden con los segmentos del genoma que están siendo ensamblados. El problema radica en el tiempo de ejecución que conlleva; los recorridos hamiltonianos dirigidos y no dirigidos.

2.3.2.3. Grafos de De Bruijn

Aunque no fueron diseñados para ello, los grafos de De Bruijn son empleados para abordar el problema que supone el ensamblaje *de novo*, siendo su mayor aportación el ensamblaje de lecturas cortas obtenidas con el uso de técnicas de secuenciación como *Solexa/Illumina* o *SOLiD*. A diferencia de los grafos de solapamiento-disposición-consenso, el recorrido del grafo es euleriano, lo que corrige el problema del tiempo de ejecución para recorridos hamiltonianos (Li *et al.,* 2011).

La representación de este tipo de grafos implica varias fases. En una primera fase se descomponen las lecturas en cadenas de longitud fija *k* denominada *k*-meros. En segundo lugar, se generan los vértices del grafo. Como se muestra en la Figura 2, los vértices corresponden con las secuencias nucleotídicas de los *k*-meros eliminando la última base de cada uno de ellos. De esta forma, los vértices tienen longitud *k*-1 y formarán prefijos y sufijos en los que solapen todas la bases excepto la última, lo que permitirá hacer el recorrido. Una vez generados los vértices, éstos son conectados por aristas que representan solapamientos perfectos de sufijo a prefijo.

También se pueden asignar orientaciones a las aristas. Se denomina grafos dirigidos a aquellos cuyas aristas conectan los vértices con una dirección definida. Los grafos dirigidos son utilizados por varios programas, tales como SOAPdenovo (Yinlong *et al.,* 2014), ABySS (Simpson *et al.,* 2009), Velvet (Zerbino *et al.,* 2008), ALLPATHS (Gnerre *et al.,* 2011) o Euler-SR (Chaisson *et al.,* 2008). En el caso de estos programas, la orientación de las aristas es única, sin embargo, puede mejorarse dado que cada *k*-mero tiene dos orientaciones, una con respecto a cada uno de sus extremos (prefijo-sufijo). Cuando el grafo se representa de esta manera se denomina grafo bidirigido. El hecho de que el grafo sea bidirigido es mejor en los casos de cadenas bicatenarias puesto que podemos escoger entre un vértice y el de la secuencia complementaria (la de menor valor léxico), reduciendo el número total de vértices. Cuando se seleccionan *k*-meros de esta manera, estos pasan a llamarse *k*-meros canónicos.

La ventaja del uso de grafos de De Bruijn es que no se requiere de elecciones durante el proceso de elaboración del grafo. En los dos métodos anteriores (algoritmos voraces y de solapamiento-disposición-consenso) se escogía la mejor entre dos o más opciones, seleccionando el mejor alineamiento posible en cada caso. Además, por su construcción, el recorrido del grafo de De Bruijn corresponde con la secuencia original,

aunque si se encuentran bucles (convergencia en el recorrido debido a la repetición de dos *k*-meros con la misma secuencia), es difícil establecer el camino. Las lecturas con solapamientos perfectos inducen un recorrido común, por ello, los solapamientos perfectos se detectan implícitamente sin necesidad de cálculos de alineamientos de secuencias emparejadas. Comparados con los grafos de solapamiento (overlap-layout-consensus), los grafos de De Bruijn son más adecuados para repeticiones o errores de secuenciación. Tras la elaboración inicial del grafo, se aplican modificaciones y simplificaciones uniendo vértices, eliminando bucles y seleccionando los mejores caminos hasta obtener un grafo irreducible del que se obtienen los cóntigos.

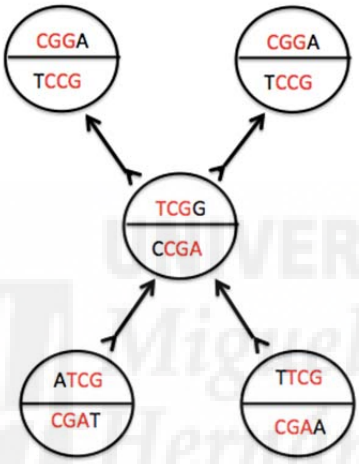

Figura 2. Ejemplo de grafo birigido de De Bruijn.

2.4. *k*-meros

Como se ha explicado en el apartado anterior, para la representación de grafos de De Bruijn es necesario la fragmentación de las lecturas en *k*-meros. Un problema que tienen estos tipo de grafos es la complejidad para la interpretación del recorrido debido a la presencia de bucles o a errores de secuenciación, lo que puede conducir a que exista más de un recorrido correcto. Con respecto a los errores de secuenciación, hay que tener en cuenta que una permutación en una base conduce a falsos negativos en el grafo, dando lugar a vértices erróneos. Así mismo, cada vértice erróneo tiene la posibilidad de coincidir con algún otro vértice y, por lo tanto, inducir a una falsa convergencia en el recorrido. Una posible solución a este problema, bajo la suposición de que los vértices sin mutaciones aparezcan con mayor frecuencia, es el recuento de *k*-meros, de tal forma que se seleccionen aquellos que aparecen un mayor número de veces, descartando así posibles errores de secuenciación.

La elección de la longitud del *k*-mero es empírica, no obstante, escoger entre una longitud corta o larga tiene efectos muy diversos en el ensamblaje. Longitudes demasiado pequeñas hacen que la posibilidad de que un *k*-mero se repita será más alta, lo que conlleva un aumento en el número de bucles en el grafo que dificultan su interpretación. Por el contrario, longitudes demasiado largas provocan que dos *k*-meros que se diferencien por un solo nucleótido debido a fenómenos de sustitución se cuenten por separado. Esto aumenta el número de falsos negativos en el grafo y dificulta la interpretación de los resultados.

Los *k*-meros se han convertido en herramientas para multitud de aplicaciones, entre las que destacan las que utilizan filtros de Bloom y el recuento de *k*-meros.

2.4.1. Aplicaciones que utilizan filtros de Bloom

Los filtros de Bloom tienen muchas aplicaciones en Bioinformática debido a que se pueden agregar elementos o comprobar si un mismo elemento está presente más de una vez consumiendo poca memoria y tiempo. Además, las búsquedas son independientes del número de elementos, por lo que se pueden paralelizar, lo que permite agilizar aún más el procedimiento. Entre otras aplicaciones destacan:

1. Recuento de *k*-meros. Varios algoritmos de recuento de *k*-meros se basan en los filtros de Bloom. Por su importancia para este trabajo, algunos algoritmos de recuento de *k*-meros se describen en mayor detalle en el siguiente apartado.
2. Corrección de errores de *k*-meros. El programa Lighter (Song *et al.,* 2014) está diseñado para la corrección de errores de secuenciación sin necesidad de contar *k*-meros. Para ello, utiliza dos filtros de Bloom, uno de entrada que contiene una muestra de *k*-meros y el otro, de salida, donde se encuentran los *k*-meros que aparecen dos o más veces (teniendo en cuenta el problema de los falsos positivos). El programa se basa en una selección probabilística mediante una distribución de Poisson que mide la frecuencia con la que aparece un determinado *k*-mero en cierto tiempo en base a la media de ocurrencias de ese *k*-mero. De este modo, todos aquellos con una probabilidad igual o mayor a un umbral marcado por la distribución de Poisson son seleccionados. Aquellos *k*-meros que se repitan un mayor número de veces serán considerados los correctos.
3. Clasificación de ADN (Stranneheim *et al.,* 2010). Se basa en un nuevo algoritmo llamado Fast and Accurate Classification of Sequences (FACS) que utiliza la estructura compacta de los filtros Bloom para una clasificación rápida y precisa de las secuencias. La aplicación principal de este algoritmos es la detección de genomas en metagenómica.

2.4.2. Recuento de *k*-meros

En teoría, el tamaño del grafo de De Bruijn sólo depende del tamaño del genoma, incluidos los alelos polimórficos, y debe de ser independiente del número de lecturas. Sin embargo, los errores de secuenciación generan falsos negativos en el grafo, y ese aumento del número de lecturas inevitablemente provoca el aumento el tamaño del grafo de Bruijn. Con el recuento de *k*-meros se ha conseguido simplificar este problema, ejemplo de ello es programa SOAPdenovo con el que, durante el ensamblaje *de novo* del genoma humano, se consiguió reducir el número de *k*-meros a partir de lecturas cortas de 14.6 mil millones a 5.0 mil millones, corrigiendo errores antes de construir el grafo de De Bruijn (Li *et al.,* 2010). Este programa cuenta el número de ocurrencias de todos los *k*-meros en las lecturas y se elimina cualquier *k*-mero que aparezca menos de tres veces (Kelley *et al.,* 2010).

El recuento de *k*-meros nos permite filtrarlos, eliminando posibles sustituciones, deleciones o inserciones previamente a la construcción del grafo. Además, el recuento de *k*-meros permite conocer la longitud del genoma mediante algoritmos de aproximación (Li *et al.,* 2003; Hozza *et al.,* 2015).

2.4.2.1. Métodos para el recuento de *k*-meros

Existen diferentes programas para el recuento de *k*-meros. Entre ellos destacan:

El programa BFCounter (Melsted *et al.,* 2011), que implementa un algoritmo de recuento de *k*-meros que utiliza un filtro de Bloom y una tabla *hash* para determinar cuántas veces aparece cada *k*-mero. En una primera fase, el programa descompone las lecturas en *k*-meros, que se añaden al filtro de Bloom. El filtro de Bloom permite identificar todos aquellos *k*-meros que aparecen más de una vez. Estos *k*-meros repetidos se incorporan a la tabla *hash*, que asocia cada *k*-mero con el número de veces que aparece. Este número se determina en una segunda etapa, en la que las lecturas vuelven a descomponerse en *k*-meros para contabilizar únicamente los *k*-meros preseleccionados inicialmente mediante el filtro de Bloom. Según este procedimiento, todos los *k*-meros de la tabla *hash* deberían aparecer dos o más veces. Por ello, en una última fase se descartan todos aquellos *k*-meros presentes una o ninguna vez, que sólo pueden corresponder a falsos positivos del filtro de Bloom.

El programa Jellyfish (Marçais *et al.,* 2011) implementa un algoritmo de recuento de *k*-meros que puede ejecutarse en paralelo. Así mismo, el algoritmo utiliza tantas tablas *hash* como hilos de procesamiento disponibles; en las tablas se almacenan todos los *k*-meros, y éstos se ordenan y se cuentan en las tablas haciendo uso del algoritmo llamado *Compare-And-Swap* (CAS). De esta forma, si los *k*-meros son iguales, se modifica el contenido de la casilla de la tabla por un nuevo valor. Jellyfish se divide en dos fases. Durante la primera fase se descomponen las lecturas en *k*-meros y, posteriormente se van haciendo rondas

donde se escoge un *k*-mero y se localiza en las tablas. Esto permite conocer la existencia de *k*-meros repetidos en las distintas tablas o si el *k*-mero aparece por primera vez, en cuyo caso se añade a la tabla *hash* que corresponda. Este procedimiento se realiza para todos los *k*-meros. La tabla *hash* contiene valores de entrada y claves; los valores de entrada corresponden a todos los *k*-meros y, a su vez, los *k*-meros se asocian a una clave que corresponde con el número de aparición de cada uno de ellos. Durante la segunda fase, se realiza el recuento de *k*-meros repetidos recorriendo todas las claves de las tablas. En el caso de que la clave no tenga valor de entrada o que sea única se deja intacta, pero si se observa una clave idéntica más de una vez significa que el *k*-mero está repetido, por lo que se modifica la clave adicionándole 1 a su valor. En el caso de Jellyfish, todas las operaciones se realizan en las distintas tablas *hash*, y para ello se suponen dos hechos: (1) que ninguna entrada se elimina nunca y, por tanto, no se precisa mantener información sobre claves eliminadas, y (2) que para el recuento de *k*-meros el tamaño requerido podría requerir toda la memoria disponible, por lo que si la tabla está llena se escribirá en el disco duro en lugar de duplicar su tamaño en memoria RAM.

El programa DSK (Rizk *et al.,* 2013) implementa un algoritmo de recuento de *k*-meros que optimiza el uso de la memoria RAM distribuyendo los *k*-meros en diferentes archivos que se almacenan en el disco. DSK se divide en tres fases. Durante la primera fase se descomponen las lecturas en *k*-meros y se realiza la distribución de los mismos. Para ello, se calcula el número total de *k*-meros, el número de iteraciones en las que se reparten los *k*-meros según la memoria disponible del disco y el número total de particiones según la memoria RAM disponible. Durante la segunda fase se distribuyen los *k*-meros en las distintas iteraciones, y dentro de cada iteración en las diferentes particiones. Las particiones se almacenan en archivos de texto que se guardan en el disco duro. Por último, se recorren todos los *k*-meros de una partición y se van contando en una tabla *hash*. Es procedimiento se realiza para todas las particiones, de modo que se va ocupando la memoria de una única partición cada la vez.

El programa KMC (Deorowicz *et al.,* 2013), al igual que DSK, almacena los datos en el disco ahorrando memoria. KMC se divide en dos fases. Durante la primera fase se distribuyen los *k*-meros (reemplazándolo por su canónico) y se almacenan en el disco según la longitud de los prefijos que construyen el grafo. De esta manera se obtienen cientos de archivos que son comprimidos antes de guardarse. Durante la segunda fase se leen los archivos y se recorren todos los *k*-meros, que se añaden a un vector donde se realizan dos operaciones: (1) se recorre el vector contando los *k*-meros y (2) se descartan los *k*-meros que estén presentes más de una vez.

El programa MSPKmerCounter (Li *et al.,* 2013) implementa un algoritmo para el recuento de *k*-meros que se basa en un método denominado partición mínima de

subcadena (MSP) (Li *et al.,* 2013). MSP parte de la suposición de que dos *k*-meros adyacentes, al tener una gran similitud, comparten una mismo segmento de secuencia que denomina subcadena *p* donde *p*<*k*. MSPKmerCounter se divide en distintas fases interconectadas. Durante la primera fase se determinan las distintas subcadenas *p* que comparten distintos conjuntos de *k*-meros y se agrupan los *k*-meros que compartan la misma subcadena *p* en particiones llamadas minimizadores (Roberts *et al.,* 2004). El almacenamiento del *k*-mero se realiza guardando los minimizadores en el disco. Al mismo tiempo que se van almacenando los minimizadores, se insertan los *k*-meros en una tabla *hash* donde se realiza el recuento de los mismos. Finalmente la información se almacena en el disco.

El programa Turtle (Roy *et al.,* 2014) implementa un algoritmo para el recuento de *k*-meros que utiliza un filtro de Bloom para detectar aquellos *k*-meros que aparecen más de una vez, y un método que permite ordenar y comprimir los elementos de un vector denominado *Sorted And Compressed array (SAC)*. Turle se divide en dos fases. Una primera fase en la que las lecturas se fragmentan en *k*-meros que se añaden al filtro de Bloom para detectar *k*-meros duplicados. Los *k*-meros que se encuentren presentes más de una vez son almacenados en un vector. Durante la segunda fase se determina un umbral de adición de elementos en el vector y se ordenan alfabéticamente los *k*-meros; si el *k*-mero es igual al anterior, se compactan y se añade 1 al recuento. Finalmente, se produce una matriz con todos los *k*-meros ordenados y contados.

El programa KAnalyze (Audano *et al.,* 2014) implementa un algoritmo de recuento de *k*-meros. KAnalyze se divide en dos fases. Durante la primera fase se dividen las secuencias en subconjuntos para cargarlas en memoria en pequeñas cantidades, lo que permite un uso de memoria mucho menor. Durante la segunda fase se dividen las lecturas en *k*-meros y se añaden a un vector. Cuando el vector está lleno, se ordenan los *k*-meros, se cuentan y se almacenan en el disco. Es procedimiento se repite hasta que se hayan contado todos los *k*-meros.

El programa Khmer (Crusoe *et al.,* 2015) implementa un algoritmo para el recuento de *k*-meros. El algoritmo utiliza una estructura de datos probabilística denominada Count-Min Sketch (Cormode *et al.,* 2005), que se basa en un filtro Bloom y el uso de múltiples tablas *hash* para detectar los *k*-meros. Khmer se dividen en tres fases. Durante la primera fase se crea una cierta cantidad de tablas *hash* de acuerdo con la cantidad máxima de falsos positivos, establecida mediante una fórmula que determina la probabilidad de falsos positivos debido a las colisiones de las funciones *hash*. En la segunda fase se verifica si un *k*-mero aparece más de una vez añadiéndolo a un filtro de Bloom. Por último, se añaden los *k*-meros a las tablas *hash* donde se realiza el recuento.

El programa KMC2 (Deorowicz *et al.,* 2015) se basa en KMC, explicado anteriormente, aunque para un mejor aprovechamiento de la memoria se utilizan minimizadores (Roberts, 2004). No obstante, en KMC2 los minimizadores han de cumplir una serie de condiciones: (1) no con AAA, (2) no empezar con ACA y (3) que no contengan AA en ninguna parte (excepto desde el principio), ya que el compartimiento asociado con el minimizador AA es generalmente enorme. Los minimizadores que cumplan estas características se llaman firmas. KMC2 se divide en dos fases. Durante la primera fase se cargan las lecturas para encontrar las firmas y almacenarlas en la misma partición del disco. Durante la segunda fase se lee cada archivo y se realiza el recuento de los *k*-meros de la misma forma que KMC. Finalmente, se guardan los resultados en una base de datos de *k*-meros.

El programa KCMBT (Mamun *et al.,* 2016) implementa un algoritmo para el recuento de *k*-meros. El algoritmo utiliza una estructura de datos tipo árbol llamado *burst tree*, que permite almacenar un conjunto de *k*-meros de manera eficiente y ordenada. Este algoritmo consta de dos fases. En una primera fase se seleccionan los *k*-meros canónicos. La información de los vértices son *k*-meros extendidos, que son subcadenas de longitud fija *k*+x, donde x es el contador (inicialmente x = 0). Si el *k*-mero sigue la misma dirección que el anterior, x= x + 1. En la primera fase se recorren los vértices contando los *k*-meros idénticos y almacenando el valor en una variable x, y en la segunda fase se dividen los *k*-meros cuyo valor de x sea el mismo y los inserta en árboles cuyo prefijo sea similar. Por último se guardan los *k*-meros y el número que aparece en el disco.

La mayoría de estos algoritmos se pueden paralelizar, es decir, se pueden ejecutar al mismo tiempo usando varios hilos de procesamiento del procesador, acortando de esta manera el tiempo de ejecución.

3. Antecedentes y Objetivos

Las nuevas tecnologías de secuenciación masivamente paralela han mejorado la secuenciación en tiempo y costes, pero producen lecturas más cortas. Esto implica una mayor cantidad de datos a tratar y, por tanto, un análisis bioinformático más avanzado por la cantidad de datos.

En el Trabajo de Fin de Grado titulado “Implementación de un programa en lenguaje Perl para el ensamblaje de secuencias nucleotídicas”, realizado en el laboratorio del Dr. Héctor Candela, ya se diseñó un programa capaz de representar grafos bidirigidos de Bruijn. El objetivo principal de este Trabajo de Fin de Grado ha sido implementar algoritmos de recuento de k-meros para mejorar el programa anteriormente citado. Estos algoritmos permiten realizar un filtro de los *k*-meros, eliminando falsos negativos del grafo antes de construirlo. Para comprobar las mejoras realizadas, se han simulado casos de ensamblaje de secuencias obtenidas en bases de datos, y se ha ido modificando diferentes parámetros como la cobertura de las secuencias o el tamaño del *k*-mero, obteniéndose cóntigos de mayor tamaño. Este objetivo ha sido concretado en los siguientes objetivos particulares:

1. Implementar el algoritmo de BFCounter en lenguaje Perl, que permitirá el recuento de *k*-meros basados en filtros de Bloom aprovechando el espacio y el tiempo de ejecución.
2. Implementar el algoritmo de DSK en lenguaje Perl para reducir el uso de la memoria a costa de un mayor consumo de tiempo.
3. Implementar el algoritmo de Turtle en lenguaje Perl para ordenar y compactar los *k*-meros almacenados dentro de un vector con el propósito de mejorar en costes de memoria y tiempo de ejecución.
4. Evaluar los efectos del recuento de *k*-meros en el ensamblaje *de novo*. Se pretende comparar el ensamblaje de diferentes secuencias obtenidas a partir de bases de datos.

4. Materiales y Métodos

4.1. Lenguaje de programación

Perl es un lenguaje de programación que fue creado por Larry Wall en 1987 para simplificar tareas de administración. La incorporación de objetos, referencias y módulos lo han convertido en un lenguaje de programación muy utilizado, siendo una de sus aplicaciones más importantes el desarrollo herramientas bioinformáticas (Tisdall, 2001).

Perl es un lenguaje interpretado, en el que las instrucciones dadas por el programador pueden ser ejecutadas por el ordenador sin necesidad de leer y traducir exhaustivamente todo el código. Así mismo, dispone de una gran cantidad de módulos para el desarrollo de casi cualquier tipo de aplicación. La disponibilidad de módulos y bibliotecas implica que existen numerosas funciones y algoritmos implementados que facilitan el trabajo del programador.

Hemos escogido este lenguaje de programación ya que es relativamente fácil de aprender y existen numerosas aplicaciones y bibliotecas, para su uso en Bioinformática como Bioperl. Hemos evaluado el código de nuestro programa mediante la herramienta Perl Critic (http://perlcritic.com), que advierte sobre posibles errores en el mismo. Para ello, hemos analizado el código con niveles de severidad comprendidos entre *gentle* (menos estricto) y *brutal* (más estricto).

4.2. Módulos

Los módulos son colecciones de subrutinas que permiten implementar funciones y algoritmos que no se encuentran por defecto disponibles para el programador. Los módulos completos o, alternativamente, algunas funciones de los mismos, pueden importarse mediante el comando *use*. La utilización de módulos permite reutilizar código en distintos programas. En este trabajo, hemos utilizado los módulos de dominio público que se describen brevemente en las siguientes secciones:

4.2.1. Bio::SeqIO

El módulo Bio::SeqIO pertenece al paquete BioPerl, que incluye numerosos módulos de aplicación a la bioinformática. Este módulo permite implementar objetos de clase Bio::SeqIO, que se utilizan para importar y exportar secuencias biológicas en los formatos más frecuentes, como fasta y fastq. Las funciones asociadas a los objetos de esta clase permiten leer las secuencia almacenadas en archivos y almacenarlas en objetos de clase Bio::Seq, que facilitan la manipulación secuencias nucleotídicas y de proteínas.

4.2.2. Bit::Vector

El módulo Bit::Vector permite crear y manipular vectores de *bit*. Con el objetivo de minimizar el consumo de memoria, nuestro programa representa cada nucleótido mediante

2 *bits*. La utilización de objetos de tipo Bit::Vector nos ha permitido almacenar secuencias de longitud superior a 32 nucleótidos, que es la longitud máxima que podría representarse en un ordenador por medio de un número entero de 64 bits.

4.2.3. Getopt::Long

El módulo Getopt::Long contiene la función `GetOptions`, que permite al usuario asignar valores a los distintos argumentos del programa desde la línea de comandos. Esta función examina la línea de comandos y almacena las opciones especificadas en variables utilizadas por el programa. En nuestro programa, hemos es posible especificar los siguientes parámetros:

`--k`	Longitud del *k*-mero
`--infile`	Permite seleccionar el archivo de entrada indicando el nombre
`--output`	Nombre del archivo de salida
`--format`	Formato de secuencia que se quiera manejar.
`--counter`	Algoritmo de recuento de k-meros a utilizar
`--mincov`	Cobertura mínima de los *k*-meros incorporados al grafo de De Bruijn
`--maxcov`	Cobertura mínima de los *k*-meros incorporados al grafo de De Bruijn

4.2.4. Bloom::Faster

El módulo Bloom::Faster permite implementar fácilmente filtros de Bloom, que hemos utilizado en varios algoritmos de recuento de *k*-meros incluidos en este programa. Como se ha descrito en la Introducción, los filtros de Bloom almacenar elementos con un consumo mínimo de memoria, por lo que han sido empleados para identificar *k*-meros duplicados con gran eficacia.

4.2.5. List::Util

Entre otras muchas, el módulo List::Util incluye una función para seleccionar el valor máximo (`max`) de un conjunto de datos. Dado que el lenguaje Perl no incluye por defecto una función para esta tarea, la hemos importado de este módulo.

4.3. Pragmas

Los pragmas son un tipo de módulo especial que permiten especificar el comportamiento del compilador. Los pragmas utilizados por nuestro programa se describen a continuación.

4.3.1. strict

El pragma `strict` permite deshabilitar ciertas expresiones de Perl que podrían comportarse inesperadamente convirtiéndolas en errores. El efecto de este pragma se limita al bloque actual del archivo o del alcance.

4.3.2. warnings

El pragma `warnings` funciona de manera similar a `strict`. De tal forma que, al ejecutar el progama, adiverte de posibles errores en el código que impiden una correcta ejecución.

4.3.3. autodie

El pragma `autodie` proporciona una manera conveniente de detener funciones que devuelven valores erróneos. De esta manera, si una función no está desempeñando la acción correctamente, autodie detiene dicha función.

4.4. Subrutinas

Las subrutinas son segmentos de código que están separados el código principal pero que realizan una función determinada que es necesaria para la correcta ejecución del código principal. El uso de subrutinas evita que el mismo segmento de código haya de ser escrito más de una vez, lo que podría causar errores de reconocimiento de variables u otros problemas durante la ejecución del programa. Las subrutinas implementadas han sido almacenadas en un módulo denominado, genéricamente, "librería", a la que se accede desde los distintos programas mediante el comando `use`. Entre las subrutinas incluidas en el programa, destacan las que permiten implementar algoritmos de recuento de *k*-meros, que se describen en el apartado 5.4.9.

4.4.1. revcom

Esta subrutina toma como argumento una secuencia nucleotídica y devuelve la secuencia complementaria. Para ello, primero intercambia cada nucleótido por su complementario y a continuación invierte el orden de los mismos.

4.4.2. to_binary

Para minimizar el consumo de memoria, la secuencia nucleotídica de los k-meros se representa mediante una secuencia de *bits*. La función `to_binary` toma como argumento una secuencia nucleotídica y devuelve una secuencia binaria (compuesta por únicamente dos tipos de caracteres, 0 y 1), según las siguientes equivalencias: A → 00, T → 11, G → 10, y C →01.

4.4.3. to_number

Transforma una secuencia nucleotídica a un número para posibilitar la conversión a binario de los datos almacenados en memoria.

4.4.4. to_nucleotides

La función `to_nucleotides` toma como argumento una secuencia binaria y devuelve una secuencia nucleotídica. Es, por tanto, la función recíproca de `to_binary`, ya que

permite convertir una secuencia representada mediante *bits* a otra representada por los cuatro nucleótidos. El cambio de nucleótidos a binario se realizó para economizar memoria y el cambio de binario a nucleótidos se realizó para visualizar las secuencias resultantes.

4.4.5. beginning y end

El programa representa las secuencias mediante un grafo bidirigido de De Bruijn, en el que las aristas o flechas poseen direccionalidad en ambos extremos. Las aristas del grafo se almacenan, tanto en el disco duro como en la memoria del ordenador, de manera compacta mediante vectores de bits. La direccionalidad de las aristas se representa mediante 2 bits: el primero especifica la orientación de la arista en su lado más próximo al vértice de origen y el segundo en su lado más próximo al vértice de destino. Las funciones `beginning` y `end` informan, respectivamente, sobre dichas orientaciones. La función `beginning` devuelve 0 cuando la flecha apunta hacia el vértice de destino y 1 en caso contrario. La función `end` devuelve 1 cuando la flecha apunta hacia el vértice de destino y 0 en caso contrario.

4.4.6. next_base

A partir de la representación en bits de una arista, la función `next_base` devuelve el nucleótido que debe añadirse al vértice de origen para alcanzar el vértice de destino. Esta subrutina es utilizada por la subrutina `assemble` (véase el apartado siguiente) para determinar con qué nucleótido debe continuar el ensamblaje.

4.4.7. assemble

La subrutina `assemble` es la encargada de ensamblar las secuencias. Para ello, selecciona al azar una arista marcada como no visitada, determina la secuencia del vértice de origen y extiende la secuencia hacia el extremo 3’ determinando con la función `next_base` que nucleótido debe añadirse para alcanzar el vértice de destino. Una vez terminada esta acción marca la arista como visitada, impidiendo que se lea más veces. Este proceso se repite hasta alcanzar un vértice ya visitado o un vértice con grado superior a 2 (correspondiente a una bifurcación). La arista inicial se utiliza, a continuación, para iniciar un ensamblaje en sentido inverso, con el que se extiende el cóntigo resultante hacia el extremo 5’ hasta que se alcanza otra bifurcación o un vértice ya visitado.

Para acceder rápidamente a la información del grafo, éste se representa en la memoria del ordenador mediante una tabla *hash* compleja, donde se utilizan como claves la secuencia del vértice de origen, por una parte, e información sobre la arista (orientación de la misma en ambos extremos) y el vértice de destino, por otra. Únicamente se almacena una letra del vértice de destino, evitando representar los nucleótidos comunes a ambos vértices (ya que en un grafo de De Bruijn las secuencias de ambos vértices son idénticas en *k*-1

posiciones). Con ambas claves se accede a 2 valores: uno que registra si la arista ha sido visitada previamente, y otro igual a la multiplicidad de dicha arista (número de veces que aparece en las lecturas).

4.4.8. alternate_edge

La subrutina `alternate_edge` permite obtener la representación alternativa de una arista, pasando a considerar que el vértice de destino es el de origen y viceversa. Es utilizada por la función `assemble`, por ejemplo, para iniciar un ensamblaje en la dirección opuesta a partir de la arista seleccionada inicialmente.

4.4.9. dsk, turtle y bfcounter

Estas tres subrutinas se corresponden con tres algoritmos de recuento de *k*-meros que hemos implementado y que se describen sucintamente en la Introducción. Estos algoritmos toman como argumentos la longitud de los *k*-meros y el nombre y formato de un archivo con la secuencia de las lecturas. Como resultado, producen una tabla *hash* con las aristas y su multiplicidad (las veces que cada una de ellas aparece en las lecturas). Los algoritmos implementados difieren en la forma de obtener los resultados. El algoritmo implementado en `bfcounter` utiliza un filtro de Bloom para identificar secuencias que aparecen más de una vez. El recuento de las veces se lleva a cabo con ayuda de una tabla *hash*. El algoritmo implementado en `dsk` está concebido para realizar el recuento de los *k*-meros cuando la cantidad de memoria es el factor limitante. Para ello, procesa un subconjunto de los *k*-meros en cada iteración con ayuda de una función, que hemos denominado `hash` (véase el apartado siguiente), que permite asignar los *k*-mero a diferentes grupos según la secuencia de los mismos. El recuento de los *k*-meros también se realiza con ayuda de una tabla *hash*. Por último, a semejanza de `bfcounter`, el algoritmo implementado en `turtle` utiliza un filtro de Bloom para seleccionar aquellas secuencias que aparecen múltiples veces en las lecturas. Sin embargo, las secuencias se almacenan en un vector y el recuento se realiza mediante la compactación de los elementos del mismo.

4.4.10. hash

La función `hash` es utilizada por el algoritmo implementado en dsk para obtener un valor numérico a partir de la secuencia de los *k*-meros. Este valor numérico se determina a partir de la secuencia de 32 nucleótidos iniciales (en cuyo caso el número resultante es un entero de 64 bits) o, si $k<32$, a partir del *k*-mero completo.

4.5. Otros programas utilizados en este trabajo

4.5.1. Graphviz

Para la visualización de grafos hemos utilizado los programas del paquete Graphviz (*Graph Visualization*). Graphviz consta de varias herramientas para la representación de

grafos. Nuestro programa produce un archivo en el que la relación (solapamiento) entre los cóntigos resultantes se representa mediante el formato `dot`, que permite especificar las características de los vértices y las aristas del grafo. Hemos utilizado los programas `dot` y `neato` para buscar la disposición (*layout*) óptima de los elementos de los grafos en el espacio y obtener representaciones gráficas de los mismos en formato `pdf` o `svg`.

4.5.2. Bowtie2

Hemos utilizado el programa Bowtie2 (Langmead *et al.,* 2009) para alinear las lecturas al genoma de referencia de *Arabidopsis thaliana* de rápida y eficiente. El uso de este programa nos ha permitido seleccionar parejas de lecturas en las que cada lectura se alinea a un cóntigo distinto. Esta información podría utilizarse fácilmente para identificar cóntigos que ocupan posiciones adyacentes y que, en consecuencia, podrían orientarse y unirse para formar supercóntigos (también denominados *scaffolds*).

4.5.3. BLAST

Hemos utilizado una implementación local del programa NCBI BLAST+ versión 2.6.0 (Basic Local Alignment Search Tool; Altschul *et al.,* 1990) para alinear la secuencia de los cóntigos obtenidos a la secuencia de los genomas de Arabidpsis thaliana y Escherichia coli, con el objetivo de validad los ensamblajes resultantes.

4.6. Origen de las secuencias utilizadas en este trabajo

En este trabajo hemos utilizado datos reales producidos en experimentos de secuenciación masivamente paralela para poner a prueba el programa. Estos datos se han obtenido desde la base de datos Sequence Read Archive (SRA; Sayers, 2010) a la que se accede desde el portal del National Center for Biotechnology Information (NCBI; https://www.ncbi.nlm.nih.gov/sra/).

Para realizar el ensamblaje *de novo* del genoma de *Escherichia coli* hemos utilizado los datos depositados con el número de acceso SRR5647033, correspondiente a un experimento de secuenciación por síntesis realizado con un equipo Illumina MiSeq. En este experimento se obtuvieron lecturas emparejadas (*paired-end reads*) hasta alcanzar aproximadamente 395 Mb. Para realizar el ensamblaje *de novo* del genoma de *Arabidopsis thaliana* hemos utilizado los datos depositados con el número de acceso SRR5491303, correspondiente a un experimento de secuenciación por síntesis realizado con un equipo Illumina HiSeq 2500. En este experimento se obtuvieron lecturas emparejadas de 150 nucleótidos de longitud hasta alcanzar 12,4 Gb. La estirpe de *Arabidopsis thaliana* secuenciada había sido sometida a mutagénesis con metanosulfonato de etilo (EMS), por lo que cabe esperar que algunas lecturas deriven de secuencias portadoras de mutaciones.

5. Resultados

Los resultados obtenidos en este Trabajo de Fin de Grado son de dos tipos. En primer lugar, hemos desarrollado un programa para el ensamblaje de secuencias nucleotídicas, que hemos programado en lenguaje Perl. En segundo lugar, hemos utilizado dicho programa para realizar sendos ensamblajes de novo de los genomas completos de la bacteria *Escherichia coli* y la planta *Arabidopsis thaliana* utilizando utilizando datos reales que hemos obtenido de la base de datos SRA. Estos datos corresponden a los genomas de *Escherichia coli* y *Arabidopsis thaliana*.

5.1. Desarrollo de un programa para el ensamblaje *de novo* de secuencias nucleotídicas

Para el desarrollo de este programa, partimos de un programa desarrollado en el curso de un Trabajo de Fin de Grado anterior (Castejón-Navarro, 2016). El programa presenta importantes mejoras sobre la versión anterior que enumeramos aquí: (1) El programa ha pasado a constar de tres subprogramas que deben ejecutarse consecutivamente. El tercero de ellos ha sido desarrollado completamente en este trabajo y permite descartar los cóntigos de menor tamaño, lo que frecuentemente resulta en un incremento del tamaño de los cóntigos. (2) El primer programa, que se encarga de realizar el recuento inicial de los *k*-meros y almacenar el grafo en el disco duro, ha sido completamente remodelado. Los cambios realizados incluyen la implementación de tres algoritmos distintos de recuento de *k*-meros, que pueden ser seleccionados a voluntad por el usuario, dependiendo de la cantidad de memoria disponible en el equipo. El uso de la memoria es mucho más eficiente ya que varios de estos algoritmos se utilizan para descartar los k-meros erróneos. (3) Además de estos tres algoritmos básicos, hemos implementado 2 versiones modificadas de los mismos. En ambas versiones (turtleFast y bfcounter2) se ha modificado el código utilizado para acceder a las lecturas almacenadas en el disco duro, lo que resuta en una ejecución más rápida. En uno de ellos (bfcounter2), además, el recuento de *k*-meros se realiza por etapas, de manera similar a como sucede en la implementación de dsk.

Como resultado de la ejecución del segundo programa, se obtienen dos archivos: uno en formato FastA con la secuencia de los cóntigos y otro en formato Dot con la información necesaria para la representación del ensamblaje mediante un grafo. Dicho archivo FastA es el que, opcionalmente, puede procesarse con el tercer programa.

5.2. Ensamblaje *de novo* del genoma de *Escherichia coli*

5.2.1. Parámetros seleccionados

Para comprobar el funcionamiento correcto de nuestro programa, realizamos primero un ensamblaje de novo del genoma de *Escherichia coli* utilizando datos públicos

que obtuvimos de la base datos SRA (número de acceso SRR5647033; véase Materiales y Métodos). Los datos disponibles incluyen la secuencia de 1.680.284 lecturas emparejadas, resultado de la secuenciación de 840.142 fragmentos por ambos extremos. Las longitudes medias de las lecturas *forward* es de 235 pb (con desviación estándar σ=38,5) y la de las lecturas *reverse* es de 236 pb (con σ=38,3).

Realizamos distintas pruebas para seleccionar empíricamente los valores de algunos parámetros, como la longitud de los *k*-meros y la cobertura mínima de los mismos para ser tenidos en consideración. Los mejores ensamblajes fueron obtenidos con *k*-meros de 51 nucleótidos (*k*=51) y cobertura mínima de 13. El algoritmo de recuento de k-meros seleccionado para ello fue `bfcounter`, que utiliza filtros de Bloom y descarta todos aquellos *k*-meros que aparecen una vez única vez.

5.2.2. Calidad del ensamblaje resultante

Tras realizar el ensamblaje, examinamos las secuencias del archivo FastA para estimar la complejidad y fiabilidad de nuestro ensamblaje. La Tabla 1 contiene información acerca de los cóntigos obtenidos, como el número total de cóntigos, el tamaño de los cóntigo más grande y más pequeño y varios descriptores de la calidad del ensamblaje (los valores de N50, L50, NG50 y LG50).

Tabla 1.- Descriptores del ensamblaje de novo del genoma de *Escherichia coli*

	Ensamblaje	
	Inicial	Secundario
Número de cóntigos	3.271	1.391
Tamaño del cóntigo más grande (pb)	48.879	95.925
Tamaño del cóntigo más pequeño (pb)	52	103
Tamaño total del ensamblaje (pb)	4.953.967	4.833.601
N50 (pb)	11.072	23.561
L50	138	65
NG50 (pb)	12.021	24.366
LG50	124	61

Como refleja la Tabla 1, el ensamblaje inicial (resultado de la ejecución del segundo programa) produjo 3.271 cóntigos de tamaños comprendidos entre 48.879 pb (el más largo) y 52 pb (el más corto). Nótese que la longitud mínima de los cóntigos es 52 porque el programa utiliza las aristas (que representan secuencias de longitud *k*+1), y no los vértices, como unidad de desplazamiento a través del grafo.

El valor N50 de nuestro ensamblaje fue 11.072 pb. El valor N50 se define como la longitud del cóntigo tal que la mitad del ensamblaje está constituida por cóntigos de

longitud igual o superior a dicho valor, y el valor L50 se define como el número de cóntigos de longitud igual o superior a la del cóntigo N50. Para el caso particular de nuestro ensamblaje, la mitad de la secuencia obtenida (2,47 Mb) se reparte entre 138 cóntigos de longitud igual o superior a 11.072 pb. De manera similar se definen los valores NG50 y LG50, que toman como referencia el tamaño conocido del genoma en lugar de la longitud total de la secuencia ensamblada. Considerando un tamaño del genoma igual a 4.639.675 pb, los valores de NG50 y LG50 de nuestro ensamblaje fueron 12.021 pb y 124, respectivamente.

Como ya hemos mencionado, el cóntigo más pequeño mide 52 pb, que es tan solo un nucleótido más largo que el tamaño del *k*-mero seleccionado y sustancialmente inferior al tamaño de las lecturas. Algunos programas que utilizan estrategias similares basadas en el uso de grafos dirigidos de De Bruijn, como Velvet (Zerbino *et al.,* 2008), descartan sistemáticamente todos los cóntigos cuya longitud es igual o menor que 2*k*. Puede razonarse que los polimorfismos en heterocigosis determinan la presencia de burbujas (*bubbles*) en el grafo de De Bruijn formadas por cóntigos de longitud mínima igual a 2*k*+1. Los cóntigos de longitud inferior, sin embargo, conducen a la formación de cabos sueltos (*hanging tips*), que dificultan la reconstrucción de la secuencia del genoma y disminuyen la longitud media de los cóntigos. Por este motivo, hemos utilizado el tercer programa para reensamblar la secuencia tras descartar todos los cóntigos de longitud igual o inferior a 2*k* (102 pb). El ensamblaje resultante tuvo un valor N50 de 23.561 pb. La mitad de la secuencia obtenida (2,41 Mb) se repartió entre 65 cóntigos de longitud igual o superior a dicho valor (23,5 kb). Considerando un tamaño del genoma igual a 4.639.675 pb, los valores de NG50 y LG50 recalculados fueron 24.366 pb y 61, respectivamente.

5.2.3. Validación del ensamblaje obtenido

Para evaluar la calidad del ensamblaje, hemos realizado alineamientos de las secuencias de mayor longitud al genoma ya conocido de *Escherichia coli* FDA00011828. Para el alineamiento se hemos utilizado el programa BLAST (Altschul *et al.,* 1990). Los 138 cóntigos de mayor longitud obtenidos en nuestro ensamblaje inicial se alinearon casi a la perfección al genoma de referencia, confirmando la gran calidad de nuestro ensamblaje.

5.2.4. Grafo resultante

Se han representado dos fragmentos del grafo final del ensamblaje de *Escherichia coli* en la Figura 3. En el caso A se trata de un fragmento del genoma, en este caso al ensamblaje no se le ha aplicado filtro por tamaño de cóntigo. No obstante, en el caso B, sí se le ha aplicado este filtro. Se puede observar como en el caso A el grafo tiene más conexiones, pero los tamaños de los vértices (representados por los números de la

figura) son menores en el caso de A que en el caso de B. Por tanto, filtrar los cóntigos y reensamblarlos permite reducir el número de vértices y aumentar el tamaño de los mismos.

A B

Figura 3. Fragmento de *k*-meros resultantes del ensamblaje de *E. coli*

5.3. Ensamblaje *de novo* del genoma de *Arabidopsis thaliana*

5.3.1. Parámetros seleccionados

Para evaluar si nuestro programa es capaz de ensamblar el genoma complejo de un organismo eucariota, realizamos también el ensamblaje *de novo* del genoma de la planta *Arabidopsis thaliana* utilizando datos públicos obtenidos de la base datos SRA (número de acceso SRR5491303; véase Materiales y Métodos). Los datos disponibles incluyen, en este caso, la secuencia de 330.833.152 lecturas emparejadas de 150 pb, resultado de la secuenciación de 165.416.576 fragmentos por ambos extremos con un secuenciador Illumina HiSeq 2500, totalizando 12,4 Gb de secuencia obtenida.

Seguimos una aproximación similar para seleccionar empíricamente la longitud de los *k*-meros y la cobertura mínima. Los mejores ensamblajes fueron obtenidos con *k*-meros de 91 nucleótidos (*k*=91) y cobertura mínima de 8. Este valor fue estimado a partir de la gráfica de la Figura 4, representada a partir de los resultados producidos por `bfcounter2`, que fue el algoritmo de recuento de *k*-meros utilizado en esta ocasión. En esta figura se aprecia claramente que hay aproximadamente 10^8 *k*-meros distintos que están representados dos veces en el conjunto de las lecturas. Es presumible que muchos

de esos *k*-meros correspondan, junto a los *k*-meros que aparecen una sola vez (no cuantificados en la figura), a errores de secuenciación. La cobertura mínima seleccionada se corresponde con el primer mínimo relativo de la Figura 4. El primer máximo relativo debe corresponder a *k*-meros derivados de la fracción de copia única del genoma nuclear. El segundo máximo, correspondiente a k-meros que aparecen en ~250 lecturas, corresponde muy probablemente a k-meros derivados del genoma cloroplástico, cuya cobertura suele ser mucho más elevada que la de los genomas nuclear y mitocondrial. Para limitar el ensamblaje al genoma nuclear de *Arabidopsis thaliana*, seleccionamos un valor máximo de de cobertura igual a 42, que debe ser suficiente para excluir lecturas derivadas de los genomas organulares.

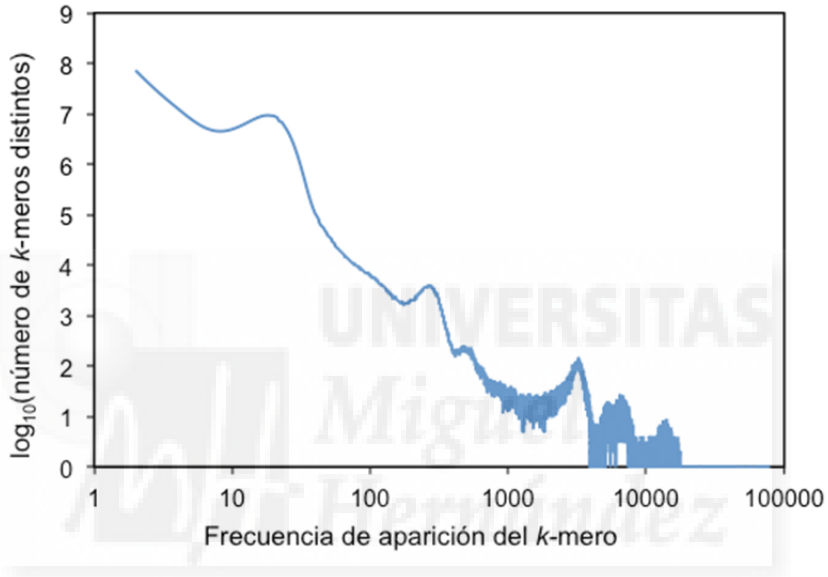

Figura 4. Número de *k*-meros (*k*=91) distintos que aparecen con una frecuencia dada en lecturas derivadas del genoma de *Arabidopsis thaliana*. Para obtener esta representación, en primer lugar se determinó la frecuencia de aparición de todos los 91-meros que aparecen 2 o más veces en las lecturas. A continuación, se determinó el número distinto de *k*-meros que presentaban la misma frecuencia de aparición (superior a 2), para cada una de las frecuencias de aparición observadas.

5.3.2. Calidad del ensamblaje resultante

Siguiendo el mismo procedimiento que para el genoma de *Escherichia coli*, examinamos las secuencias de los cóntigos para evaluar las características del ensamblaje realizado. La Tabla 2 resume algunas características de los cóntigos obtenidos, incluyendo su número total, el tamaño de los cóntigos mayor y menor y los descriptores de la calidad del ensamblaje (N50, L50, NG50 y LG50).

Tabla 2.- Descriptores del ensamblaje de novo del genoma de *Arabidopsis thaliana*

	Ensamblaje	
	Inicial	Secundario
Nº total de cóntigos	762.066	166.774
Tamaño del cóntigo más grande (pb)	28.501	28.501
Tamaño del cóntigo más pequeño (pb)	92	183
Tamaño total del ensamblaje (pb)	147.454.493	147.285.770
N50 (pb)	2.364	2.368
L50	16.087	16.051
NG50 (pb)	2.700	2.706
LG50	13.619	13.594

Como refleja la Tabla 2, el ensamblaje inicial produjo 762.066 cóntigos de tamaños comprendidos entre 92 y 28.501 nucleótidos. El valor N50 del ensamblaje fue 2.364 pb, y el valor L50 fue de 16.087. En consecuencia, la mitad de la secuencia obtenida (~73,7 Mb) se reparte entre 16.087 cóntigos de longitud igual o superior a 2.364 pb. Considerando que un genoma de 135 Mb, los valores de NG50 y LG50 de nuestro ensamblaje fueron 2.700 pb y 13.619, respectivamente.

Hemos utilizado el tercer programa para reensamblar la secuencia tras descartar todos los cóntigos de longitud igual o inferior a 2*k* (182 pb). El ensamblaje resultante tuvo un valor N50 de 2.368 pb. La mitad de la secuencia (73,6 Mb) se repartió entre 16.051 cóntigos de longitud igual o superior a 2368 pb. Los valores recalculados de NG50 y LG50 fueron 2.706 pb y 13.594, respectivamente. Como se aprecia en la Tabla 2, el número total de cóntigos se redujo por un factor de 4,5 tras este paso. No obstante, el tamaño del cóntigo más grande fue el mismo, lo que indica que los cóntigos no pudieron concatenarse para dar lugar a otros de mayor longitud en este caso particular.

5.3.3. Validación del ensamblaje obtenido

Hemos evaluado si la secuencia de los cóntigos obtenidos se corresponde con la del genoma de referencia de *Arabidopsis thaliana*. Para ello, hemos alineado los 500 cóntigos de mayor longitud con la secuencia del genoma de esta especie utilizando el programa BLAST (Altschul et al., 1990). Obtuvimos alineamientos perfectos con la secuencia de 443 cóntigos. Los 57 restantes se diferenciaron únicamente en una base del genoma de referencia. Como esperábamos, la selección de los *k*-meros con coberturas superiores a 42 excluyó de facto a las secuencias derivadas de los genomas organulares. Todos los cóntigos alineados correspondieron a los 5 cromosomas del

genoma nuclear de *Arabidopsis thaliana*, de manera proporcional a la longitud de cada cromosoma: 148 correspondieron al cromosoma 1, 72 al cromosoma 2, 103 al cromosoma 3, 81 al cromosoma 4 y 96 al cromosoma 5.

5.3.4. Ensamblaje de los cóntigos en unidades de orden superior (supercóntigos o *scaffolds*)

Dado que nuestro programa no aprovecha la información asociada a las parejas de lecturas, realizamos un experimento piloto para evaluar cómo variaría la calidad del ensamblaje si dicha información fuese tenida en cuenta. Para identificar cóntigos adyacentes, realizamos el alineamiento de las lecturas a los cóntigos obtenidos mediante el programa Bowtie2 (Langmead *et al*., 2009; véase Materiales y Métodos). El examen cuidadoso del archivo resultante en formato SAM, que contiene una descripción del alineamiento de las lecturas a los distintos cóntigos, nos permitió seleccionar las parejas en las que cada lectura se alineaba a un cóntigo diferente. Tras tabular el número de fragmentos que se alineaban al mismo par de cóntigos. Tras aplicar un valor umbral de 5 fragmentos, hemos visto que los pares de cóntigos que exceden ese valor ocupan posiciones adyacentes en el cromosoma 5 de *Arabidopsis thaliana*, como hemos determinado para los 12 cóntigos de la Tabla 3, que recoge las coordenadas de los cóntigos determinadas mediante alineamientos realizados con el programa BLASTn.

Tabla 3.- Identificación de cóntigos adyacentes para definir supercóntigos o *scaffolds*

		Coordenada (cóntigo)		Coordenada (cromosoma 5)	
Cóntigo	Tamaño (pb)	Inicial	Final	Inicial	Final
132.823	389	1	389	24.055.266	24.055.654
88.678	215	1	215	24.055.752	24.055.966
65.982	608	608	1	24.055.883	24.056.490
17.283	15.478	1	15.478	24.056.422	24.071.899
39.056	760	760	1	24.071.810	24.072.569
70.984	832	1	832	24.072.488	24.073.319
17.198	5.583	1	5.583	24.073.266	24.078.848
40.008	3.045	1	3.045	24.078.760	24.081.804
54.403	1.119	1	1.119	24.081.715	24.082.833
35.259	2.589	2.589	1	24.082.745	24.085.333
84.196	268	1	268	24.085.244	24.085.511
87.315	322	322	1	24085478	24085799

6. Discusión

En este trabajo hemos realizado importantes mejoras a un programa que realiza ensamblajes *de novo* de secuencias nucleotídicas utilizando una estrategia basada en el uso de grafos bidirigidos de De Bruijn. Con respecto a la versión anterior del programa, la principal mejora consiste en la implementación en lenguaje de programación Perl de varios algoritmos de recuento de *k*-meros. La implementación ha sido realizada a partir de la descripción detallada de dichos algoritmos mediante pseudocódigo obtenida de las correspondientes publicaciones científicas. Los algoritmos seleccionados están implementados en los programas BFCounter, DSK y Turtle y han sido implementados mediante tres subrutinas, que hemos denominado `bfcounter`, `dsk` y `turtle`, respectivamente. El algoritmo de recuento de k-meros a utilizar se ha incorporado como una de las opciones que el usuario puede especificar en la línea de comandos del primer programa. La posibilidad de elegir entre distintos métodos supone una ventaja, ya que el usuario puede seleccionar el algoritmo más adecuado según las características del equipamiento informático disponible. Por ejemplo, la ejecución de `dsk` requiere menos memoria, a costa de requerir un mayor tiempo de ejecución.

Esta nueva versión del programa consta de tres subprogramas que deben ejecutarse consecutivamente. Con el primer subprograma, denominado `step1`, las lecturas se fragmentan en *k*-meros, se realiza el recuento según el algoritmo seleccionado y, finalmente, se almacena el grafo de De Bruijn en el disco duro, representando la secuencia de los vértices y la direccionalidad de las aristas mediante código binario. El segundo subprograma, denominado `step2`, recupera el grafo del disco duro y lo almacena en la memoria del ordenador mediante una tabla *hash*. A continuación, realiza un recorrido por el grafo, visitando sistemáticamente todas las aristas del mismo. Los cóntigos se ensamblan bidireccionalmente, atravesando aristas consecutivas. El ensamblaje se detiene cuando se visita un vértice cuyo grado es igual o superior a 3 (se denomina grado de un vértice al número de aristas que inciden sobre el mismo), un valor que indica la existencia de una bifurcación en el grafo de De Bruijn, o cuando se alcanza un vértice sobre el que únicamente inciden aristas que han sido visitadas previamente. Los resultados producidos por este programa se almacenan en dos archivos en el disco duro: el primero contiene la secuencia de los cóntigos en formato FastA y, el segundo guarda una descripción de los solapamientos existentes entre dichos cóntigos en formato estándar Dot, utilizado por el paquete Graphviz. Los archivos en este formato pueden utilizarse para representar el ensamblaje mediante un grafo bidirigido en el que cada cóntigo se representa mediante una arista y los solapamientos entre los mismos (de longitud *k*) se representen mediante vértices.

Adicionalmente, esta nueva versión del programa incorpora un tercer subprograma, denominado `step3`, que permite mejorar sustancialmente el resultado final. Para ello, el programa descarta todos aquellos cóntigos (leídos a partir del archivo FastA producido por `step2`) cuya longitud es igual o inferior al doble del valor de *k* utilizado en el ensamblaje. Este filtro excluye de facto a los cabos sueltos (hanging tips) que con gran probabilidad corresponden a errores de secuenciación presentes en las lecturas. Tras la selección de los cóntigos de tamaño superior a 2*k*, el programa realiza un ensamblaje secundario, concatenando aquellos cóntigos que definen recorridos a través del grafo sin ambigüedad.

Además de los tres programas descritos (`step1`, `step2` y `step3`) hemos ampliado la batería de funciones y algoritmos que forman parte del archivo denominado `libreria`, en el que se guarda la colección de subrutinas utilizadas por el programa. Las nuevas subrutinas incorporadas a la colección incluyen: (a) las correspondientes a los cinco algoritmos de recuento de *k*-meros descritos (tres básicos y dos variantes), (b) la subrutina denominada `assemble_fasta`, responsable del ensamblaje secundario de las secuencias del archivo FastA llevado a cabo por el programa `step3`, y (c) una subrutina denominada genéricamente `hash`, que puede utilizarse para traducir las secuencias a código binario y utilizar las secuencias en dicho código para realizar el recuento. De esta manera, se aumenta la capacidad de memoria. `bfcounter` es una de las subrutinas implementadas. Este algoritmo se basa en los filtros de Bloom y las tablas *hash* para el recuento de *k*-meros puesto que es una manera fácil y que consume poca memoria, en primer lugar, detecta la existencia de *k*-meros duplicados mediante el filtro de Bloom y, posteriormente, realiza el recuento en una tabla *hash*. `dsk` es otro de los algoritmos para el recuento de *k*-meros implementados, la estrategia de este algoritmo es realizar particiones en el disco en las que se almacenan conjuntos de *k*-meros, a continuación, se leen y se cuentan en una tabla *hash*. Hemos realizado una mejora de este algoritmo con respecto al utilizado por el programa original DSK que es el uso de *k*-meros canónicos, disminuyendo a la mitad los datos a procesar. `turtle` es el último algoritmo que hemos implementado, se parece a `bfcounter` puesto que también se basa en filtros de Bloom para determinar la presencia de duplicados de *k*-meros, no obstante, en este caso el recuento se realiza utilizando un vector y ordenando y modificando sus elementos. A este algoritmo también se le ha aplicado la mejora de los *k*-meros canónicos. Además se ha modificado la forma de leer los datos de tal forma que agilice el proceso dando lugar a otras dos subrutinas `bfcounter2` y `turtleFast`. Estas subrutinas leen los datos directamente desde el archivo donde se encuentren las lecturas sin tener que tratarlas como objetos de secuencia. Esto permite dismuir el tiempo de ejecución. En el caso de

`bfcounter2` también hemos modificado la manera de almacenar los datos en memoria, en lugar de almacenar todos los *k*-meros al mismo tiempo se distribuyen previamente en iteraciones, cargando paquetes de *k*-meros.

Para comprobar la eficiencia de cada programa, hemos puesto a prueba el programa y las nuevas implementaciones ensamblando datos reales de genomas de *Escherichia coli* y *Arabidopsis thaliana* con números de acceso SRR5647033 SRR5491303, respectivamente (véase Materiales y Métodos). Se ha modificado el programa de recuento, el tamaño de *k*-mero y número de frecuencia de *k*-mero que se quiere seleccionar. El resultado es que se obtienen cóntigos más largos si se compara el programa anterior (Castejón-Navarro, 2016) con el actual. Este nuevo programa permite el procesamiento de una mayor cantidad de datos, antes se realizaban ensamblajes de 150kb y ahora se realizan ensamblajes de 12,4 Gpb. Por tanto, el recuento de *k*-meros es una manera eficaz para representar genomas de gran tamaño.

Por otro lado, se ha evaluado el ensamblaje con diferentes parámetros. Dichos parámetros han sido: el número total de cóntigos obtenidos, el tamaño del cóntigo más grande y el del más pequeño y valores de N50, L50, NG50 y LG50. Estos parámetros han sido medidos para el ensamblaje inicial y para un ensamblaje en el cual se han seleccionado los cóntigos según su tamaño. Se han escogido aquellos cóntigos cuyo tamaño fuese mayor a 2*k* y, posteriormente, los cóntigos resultantes se han vuelto a ensamblar, compactándolos y reduciendo el número de vértices a representar. Para ambos ensamblajes los resultados fueron satisfactorios aunque en el caso del *Arabidopsis thaliana* no se consiguió compactar los vértices. Esto puede deberse a que como se comprobó realizando el *scaffolding* el ensamblaje no era perfecto debido a que faltaban varios nucleótidos para conectar los vértices. El scaffolding se realizó con los datos del ensamblaje inicial lo que implica que el filtro de los *k*-meros fue muy severo descartando posibles secuencias de cobertura menor a 7 importantes para el correcto ensamblaje.

Se ha realizado una segunda comprobación del ensamblaje evaluando si los cóntigos obtenidos correspondían a los genomas descargados. Para ello, se utilizó el programa BLAST, el cual, se encargó de alinear los cóntigos en una base de datos. Los alineamientos entre secuencias dieron altos valores de similud lo que demuestra que los cóntigos pertenecen a las especies que se esperaban (*Escherichia coli* y *Arabidopsis thaliana*). Para realizar esta comprobación se han utilizado los datos sin filtrado de cóntigos, intentado de esta forma, realizar el menor número de aproximaciones posibles.

En el caso de *Escherichia coli* se han representado dos fragmentos del grafo final. Un fragmento del ensamblaje inicial y un segundo fragmento del ensamblaje tras la

ejecución de `step3`. Comprobando así que se conseguía compactar los vértices, facilitando la interpretación del grafo final.

7. Conclusiones y proyección futura

Este trabajo de Fin de Grado surge como continuación de un programa elaborado anteriormente para el ensamblaje *de novo* basado en grafos bidirigidos de De Bruijn. A dicho programa se le han implementado mejoras, entre ellas, el recuento de *k*-meros que permite filtrarlos. De esta manera, se seleccionan aquellos k-meros que aparecen más veces y, debido a ello, suponemos que no contienen errores de secuenciación. El proceso de selección de k-meros es previo a la construcción del grafo lo que aumenta la capacidad de memoria y permite la construcción de cóntigos más grandes.

Hemos aplicado el programa y las mejoras implementadas al ensamblaje de los genomas de *Escherichia coli* y *Arabidopsis thaliana*. En estos ensamblajes hemos utilizado diferentes parámetros, que hemos modificado en diferentes pruebas para ver los efectos que provocaban. Los parámetros o variables han sido: la cobertura, el tamaño del *k*-mero y el rango de filtro que está limitado por el número de repeticiones mínimas y máximas de *k*-meros que se quiere seleccionar.

Hemos evaluado el ensamblaje utilizando diferentes parámetros, como el número total de cóntigos obtenidos, el tamaño del cóntigo más grande y el del más pequeño y valores de N50, L50, NG50 y LG50.

Así mismo, hemos comprobado que los cóntigos obtenidos correspondían con las secuencias esperadas realizando un blast.

En cuanto a la proyección futura, pretendemos optimizar aún más el uso de memoria RAM para que sea posible ensamblar genomas de mayor tamaño en menor tiempo. Este objetivo se podría solucionar mediante dos vías. Una de las formas para reducir la memoria utilizada puede ser la corrección de errores de las lecturas (Li, 2015) lo que disminuiría la cantidad de datos a procesar. La otra forma, sería cambiando la forma en la que se tratan los datos durante la subrutina `assembler`. Actualmente esta subrutina utiliza los datos del archivo Dot, el cual, contiene el vértice de origen, el vértice de destino, la arista y la cobertura, por tanto, a la hora de tratamiento de datos, se compara los vértices de origen y destino. En lugar de ello, una posible mejora es almacenar en el archivo dot solo los vértices de origen y la última letra del vértice de destino, ahorrando así almacenar en la memoria todas las bases que se repiten en el solapamiento. Una tercera manera de conseguir el objetivo sería si se representase el grafo mediante un filtro de Bloom (Chikhi *et al.*, 2013).

8. Bibliografía

Alkan, C., Sajjadian, S. y Eichler, E.E. (2011). Limitations of next-generation genome sequence assembly. *Nat. Methods* **8:** 61-65.

Altschul, S.F., Gish, W., Miller, W., Myers, E.W. y Lipman, D.J. (1990). Basic local alignment search tool. *J.Mol.Biol.* **5:** 403-410.

Audano, P. y Vannberg, F. (2014). KAnalyze: a fast versatile pipelined k-mer toolkit. *Bioinformatics* **30:** 2070-2072.

Bloom, B.H. (1970). Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **13:** 422-426.

Chaisson, M.J. y Pevzner, P.A. (2008). Short read fragment assembly of bacterial genomes. *Genome Res.* **18:** 324-330.

Chikhi, R. y Rizk, G. (2013). Space-efficient and exact de Bruijn graph representation based on a Bloom filter. *Algorithms Mol. Biol.* **8:** 22

Chu, J., Sadeghi, S., Raymond, A., Jackman, S.D., Nip, K.M., Mar. R., Mohamadi. H., Butterfield, Y.S., Robertson, A.G. y Birol. I. (2014). BioBloom tools: fast, accurate and memory-efficient host species sequence screening using bloom filters. *Bioinformatics* **30:** 3402-3404.

Cormode, G. y Muthukrishnan, S. (2005). An Improved Data Stream Summary: The Count-Min Sketch and its applications. *J. Algorithms* **55:** 58-75.

Crusoe, M.R., Edvenson, G., Fish, J., Howe, A., McDonald, E., Nahum, J., Nanlohy, K., Ortiz-Zuazaga, H., Pell, J., Simpson, J., Scott, C., Srinivasan, R.R., Zhang, Q. y Brown, C.T. (2015). The khmer software package: enabling efficient nucleotide sequence analysis. *F1000 Res*. **4:** 900.

De la Bastide, M. y McCombie, W.R. (2007). Assembling Genomic DNA Sequences with PHRAP. *Curr. Protoc. Bioinformatics* Chapter 11: Unit11.4.

Deorowicz, S., Debudaj-Grabysz, A. y Grabowski, S. (2013). Disk-based *k*-mer counting on a PC. *BMC Bioinformatics* **14:** 160.

Deorowicz, S., Kokot, M., Grabowski, S. y Debudaj-Grabysz, A. (2015). KMC 2: fast and resource-frugal *k*-mer counting. *Bioinfomatics* **31:** 1569-1576.

Dressman, D., Yan, H., Traverso, G., Kinzler, K.W. y Vogelstein, B. (2003). Transforming single DNA molecules into fluorescent magnetic particles for detection and enumeration of genetic variations. *Proc. Natl. Acad. Sci. USA* **100:** 8817-8822.

Edgar, R.C. y Flyvbjerg, H. (2015). Error filtering, pair assembly and error correction for next-generation sequencing reads. *Bioinformatics* **31:** 3476-3482.

Ekblom, R. y Wolf, J.B. (2014). A field guide to whole-genome sequencing, assembly and annotation. *Evol. Appl.* **7:** 1026-1104.

El-Metwally, S., Hamza, T., Zakaria, M. y Helmy, M. (2013). Next-generation sequence assembly: four stages of data processing and computational challenges. *PLoS Comput. Biol.* **9:** e1003345.

Fedurco, M., Romieu, A., Williams, S., Lawrence, I. y Turcatti, G. (2006). BTA, a novel reagent for DNA attachment on glass and efficient generation of solid-phase amplified DNA colonies. *Nucleic Acids. Res.* **34:** e22.

Gnerre, S., Maccallum, I., Przybylski, D., Ribeiro, F.J., Burton, J.N., Walker, B.J., Sharpe, T., Hall, G., Shea, T.P., Sykes, S., Berlin, A.M., Aird, D., Costello, M., Daza, R., Williams, L., Nicol, R., Gnirke, A., Nusbaum, C., Lander, E.S. y Jaffe, D.B. (2011). High-quality draft assemblies of mammalian genomes from massively parallel sequence data. *Proc. Natl. Acad. Sci. USA* **108:** 1513-1518.

Hozza, M., Vinar, T. y Brejová, B. (2015). How big is that genome? Estimating genome size and coverage from *k*-mer abundance spectra. *String Processing and Information Retrieval* **9309:** 199-209.

Joyanes-Aguilar, L. (2008). Estructura de datos. *Fundamentos de Programación, Algoritmos, estructura de datos y objetos* (4ª ed.). McGraw-Hill.

Kelley, D.R., Schatz, M.C. y Salzberg, S.L. (2010). Quake: quality-aware detection and correction of sequencing errors. *Genome Biol*. **11:** R116.

Langmead, B., Trapnell, C., Pop, M. y Salzberg, S.L. (2009). Ultrafast and memory-efficient alignment of short DNA sequences to the human genome. *Genome Biol*. **10:** R25.

Li, H. (2015). Correcting Illumina sequencing errors for human data. *Bioinformatics* **31:** 2885-2887.

Li, H. y Durbin, R. (2009). Fast and accurate short read alignment with Burrows-Wheeler transform. *Bioinformatics* **25:** 1754-1760.

Li, R., Yu, C., Li, Y., Lam, T.W., Yiu, S.M., Kristiansen, K. y Wang, J. (2009). SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics* **25:** 1966-1967.

Li, R., Zhu, H., Ruan, J., Qian, W., Fang, X., Shi, Z., Li, Y., Li, S., Shan, G., Kristiansen, K., Li, S., Yang, H., Wang, J. y Wang, J. (2010). De novo assembly of human genomes with massively parallel short read sequencing. *Genome Res*. **2:** 265-272.

Li, X. y Waterman, M.S. (2003). Estimating the repeat structure and length of DNA sequences using Q-tuples. *Genome Res.* **13:** 1916-1922.

Li, Y. y Yan, X. (2013). MSPKmerCounter: A fast and memory efficient approach for *k*-mer counting. *arXiv*:1505.06550

Li, Y., Kamousi, P., Han, F., Yang, S., Yan, X. y Subhash, S. (2013). Memory efficient minimum substring partitioning. Proceedings VLDB Endowment. **6:** 169-180.

Li, Z., Chen, Y., Mu, D., Yuan, J., Shi, Y., Zhang, H., Gan, J., Li, N., Hu, X., Liu, B., Yang, B. y Fan, W. (2011). Comparison of the two major classes of assembly algorithms: overlap-layout-consensus and de-Bruijn-graph. *Brief. Funct. Genomics* **11:** 25-37.

Mamun, A., Pal, S. y Rajasekaran, S. (2016). KCMBT: a *k*-mer counter based on multiple burst trees. *Bioinformatics* **32:** 2783-2790.

Marçais, G. y Kingsford, C. (2011). A fast, lock-free approach for efficient parallel counting of occurrences of *k*-mers. *Bioinformatics* **27:** 764-770.

Medvedev, P., Georgiou, K., Myers, G. y Brudno, M. (2007). Computability of models for sequence assembly. *Lecture Notes in Bioinformatics* **4645:** 289-301.

Melsted, P. y Pritchard, J.K. (2011). Efficient counting of *k*-mers in DNA sequences using a Bloom filter. *BMC Bioinformatics* **12:** 333.

Miller, J.R., Koren, S. y Sutton, G. (2010). Assembly algorithms for next-generation sequencing data. *Genomics* **95:** 315-327.

Nagarajan, N. y Pop, M. *(2013).* Sequence assembly demystified. *Nat. Rev. Genet.* **14:** 157-167.

Pevzner, P.A., Tang, H. y Waterman, M.S. (2001). An Eulerian path approach to DNA fragment assembly. *Proc. Natl. Acad. Sci. USA* **98:** 9748-9753.

Rizk, G., Lavenier, D. y Chikhi, R. (2013). DSK: *k*-mer counting with very low memory usage. *Bioinformatics* **29:** 652-653.

Roberts, M., Hunt, B.R., Yorke, J.A., Bolanos, R.A., y Delcher, A.L. (2004). A preprocessor for shotgun assembly of large genomes. *J. Comput. Biol.* **11:** 734-752.

Ronaghi, M., Uhlén, M. y Nyrén, P. (1998). A sequencing method based on real-time pyrophosphate. *Science* **281:** 363-365.

Rothberg, J.M., Hinz, W., Rearick, T.M., Schultz, J., Mileski, W., Davey, M., Leamon, J.H., Johnson, K., Milgrew, M.J., Edwards, M., Hoon, J., Simons, J.F., Marran, D., Myers, J.W., Davidson, J.F., Branting, A., Nobile, J.R., Puc, B.P., Light, D., Clark, T.A., Huber, M., Branciforte, J.T., Stoner, I.B., Cawley, S.E., Lyons, M., Fu, Y., Homer, N., Sedova, M., Miao, X., Reed, B., Sabina, J., Feierstein, E., Schorn, M., Alanjary, M., Dimalanta, E., Dressman, D., Kasinskas, R., Sokolsky, T., Fidanza, J.A., Namsaraev, E., McKernan, K.J., Williams, A., Roth, G.T. y Bustillo, J. (2011). An integrated semiconductor device enabling non-optical genome sequencing. *Nature* **475:** 348-352.

Roy, R.S., Bhattacharya, D. y Schliep, A. (2014). Turtle: Identifying frequent *k*-mers with cache-efficient algorithms. *Bioinformatics* **30:** 1950-1957.

Sanger, F., Nicklen, S. y Coulson, A.R. (1977). DNA sequencing with chain-terminating inhibitors. *Proc. Natl. Acad. Sci. USA* **74:** 5463-5467.

Sayers, E.W., Barrett, T., Benson, D.A., Bolton, E., Bryant, S.H., Canese, K., Chetvernin, V., Church, D.M., DiCuccio, M., Federhen, S., Feolo, M., Geer, L.Y., Helmberg, W., Kapustin, Y., Landsman, D., Lipman, D.J., Lu, Z., Madden, T.L., Madej, T., Maglott, D.R., Marchler-Bauer, A., Vadim Miller, V., Mizrachi, I., Ostell, J., Panchenko, A., Pruitt, K.D., Schuler, G.D., Sequeira, E., Sherry, S.T., Shumway, M., Sirotkin, K., Slotta, D., Souvorov, A., Starchenko, G., Tatusova, T.A., Wagner, L., Wang, Y., W. Wilbur W.J., Yaschenko, E. y Ye, J. (2010). Database resources of the National Center for Biotechnology Information. *Nucleic Acids Res*. **38:** D5-16

Sedgewick, R. y Wayne, K. (2011). Searching. *Algorithms* (4ª ed). Addison-Wesley.

Simpson, J.T., Wong, K., Jackman, S.D., Schein, J.E., Jones J.M. y Birol, I. (2009). ABySS: a parallel assembler for short read sequence data**.** *Genome Res.***19:** 1117-1123.

Song, L., Florea, L. y Langmead, B. (2014). Lighter: fast and memory-efficient error correction without counting. *Genome Biol.* **15:** 509.

Staden, R. (1979). A strategy of DNA sequencing employing computer programs. *Nucleic Acids Res*. **6:** 2601-2610.

Stranneheim, H., Käller, M., Allander, T., Andersson, B., Arvestad, L. y Lundeberg, J. (2010). Classification of DNA sequences using Bloom filters. *Bioinformatics* **26:** 1595-1600.

Tisdall, J. (2001). Getting Started with Perl. *Beginning Perl for Bioinformatics*. O'Reilly.

Turcatti, G., Romieu, A., Fedurco, M. y Tairi, A.P. (2008). A new class of cleavable fluorescent nucleotides: synthesis and optimization as reversible terminators for DNA sequencing by synthesis. *Nucleic Acids Res*. **36:** e25.

Venter, J.C., Adams, M.D., Myers, E.W., Li, P.W., Mural, R.J., Sutton, G.G., Smith, H.O., Yandell, M., Evans, C.A., Holt, R.A., Gocayne, J.D., Amanatides, P., Ballew, R.M., Huson, D.H., Wortman, J.R., Zhang, Q., Kodira, C.D, *et al.* (2001). The sequence of the human genome. *Science* **291:** 1304-1351.

Xu, R. y Cun-Quan,Z. (2005). On flows in bidirected graphs. *Discrete Math.* **299:** 335-343.

Yinlong, X., Gengxiong, W., Jingbo, T., Ruibang, L., Jordan, P., Shanlin, L., Weihua, H., Guangzhu, H., Shengchang, G., Shengkang, L., Xin, Z., Tak-Wah, L., Yingrui, L., Xun, X., Gane Ka-Shu, W. y Jun, W. (2014). SOAPdenovo-Trans: de novo transcriptome assembly with short RNA-Seq reads. *Bioinformatics* **30:** 1660-1666.

Zerbino, D.R. y Birney, E. (2008). Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome Res.* **18:** 821-829.

Zhang, Z., Schwartz, S., Wagner, L. y Miller, W. (2000). A greedy algorithm for aligning DNA sequences. *J. Comput. Biol*. **7:** 203-214.

9. Apéndice: Código del programa

9.1. Step1.pl

```
#!/usr/bin/env perl
package Assembler;
use strict;
use warnings;
use autodie;
use Bit::Vector;
use Bio::SeqIO;
use Getopt::Long;
use Bloom::Faster;
use List::Util qw[min max];
use libreriaFinal2;

my $infile = '';
my $outfile = '';
my $k = 35;           # default k-mer length
my $format = 'fastq'; # default input format
my $counter= 'dsk';   # default k-mer counter
my $mincov = 2;       # default minimum coverage
my $kref;

my $commandline = join ' ', $0, @ARGV;
open my $log_fh, '>>', 'assembler.log';
print {$log_fh} "$commandline\n";
close $log_fh;

if (!GetOptions ('kmer=i'=>\$k,
                 'input=s' => \$infile,
                 'output=s' =>\$outfile,
                 'format=s' =>\$format,
                 'counter=s' =>\$counter,
                 'mincov=i'=> \$mincov) or  $infile eq '' or $outfile eq '')
    {
       usage();
    }

if ($k%2 == 0) { $k++; } # k-mer length must be an odd number

if ($counter eq 'bfcounter') {  #choide of k-mer counting algorithm
     $kref = bfcounter ($infile, $format, $k+1);
   } elsif ($counter eq 'dsk'){
     $kref = dsk ($infile, $format, $k+1);
   } elsif ($counter eq 'turtle'){
     $kref = turtle ($infile, $format, $k+1);
   } elsif ($counter eq 'turtleFast'){
     $kref = turtleFast ($infile, $format, $k+1);
```

```
   } elsif ($counter eq 'bfcounter2'){
     $kref = bfcounter2 ($infile, $format, $k+1);
   }

print "K-mer counting finished\n";

open my $file_out, '>', 'output_final.bin';
binmode $file_out;

my $repeated = 2;
my $pattern = 'B' . ($k * 2 + 20);

foreach my $kmers (keys %{ $kref }) {
   if ($kref->{$kmers}< $repeated){
      delete $kref->{$kmers};
   } else {
      my $subsequence_1 = substr $kmers, 0, $k;
      my $RC_1 = revcom($subsequence_1);
      my ($print_1, $arrow1);
      if ($subsequence_1 lt $RC_1) {
         $print_1 = $subsequence_1;
         $arrow1 = '0';
      } else {
         $print_1 = $RC_1;
         $arrow1 = '1';
      }
      my $subsequence_2 = substr $kmers, 1, $k;
      my $RC_2 = revcom($subsequence_2);
      my ($print_2, $arrow2);
      if ($subsequence_2 lt $RC_2) {
         $print_2 = $subsequence_2;
         $arrow2 = '1';
      } else {
         $print_2 = $RC_2;
         $arrow2 = '0';
      }
      my $bin_1 = to_binary ($print_1);
      my $bin_2 = to_binary ($print_2);
      my ($output1, $output2);
      if ($arrow2 eq '0'){
         $output1 = $bin_1 . $arrow1 . '0' . to_binary(revcom (substr $print_2, 0 , 1));
      } else {
         $output1 = $bin_1 . $arrow1 . '1' . to_binary(substr $print_2, $k-1,1);
      }
      if ($arrow1 eq '0'){
         $output2 = $bin_2 . $arrow2 . '0' . to_binary(revcom (substr $print_1, 0 , 1));
      }else{
         $output2 = $bin_2 . $arrow2 . '1' . to_binary(substr $print_1, $k-1, 1);
      }
      if ($kref->{$kmers}>2**16-1){$kref->{$kmers}=2**16-1;}
```

```
      if ($output1 ne $output2) {
         my $number_1 = $output1 . to_number($kref->{$kmers}, 8);
         print {$file_out} pack $pattern, $number_1;
         my $number_2 = $output2 . to_number($kref->{$kmers}, 8);
         print {$file_out} pack $pattern, $number_2;
      } else {
         my $number_1 = $output1 . to_number($kref->{$kmers}, 8);
         print {$file_out} pack $pattern, $number_1;
      }
      delete $kref->{$kmers};
   }
}

undef %{ $kref };
close $file_out;

sub usage {
   print "Unknown option: @_\n" if (@_);
   print "\n$0:\n";
   print "This program prepares a bidirected de Bruijn graph from a set of read sequences.\n\n";
   print "Mandatory options:\n";
   print " --input           STRING  Fasta file with reads\n";
   print " --output          STRING  Name for intermediate files\n";
   print " --kmer            INT     k-mer length\n";
   print " --format          STRING  fasta | fastq\n";
   print " --counter         STRING  dsk|bfcounter|turtle\n";
   print " --mincov          INT     Minimum coverage\n";
   exit;
}
```

9.2. Step2.pl

```
#!/usr/bin/env perl
package Assembler;
use strict;
use warnings;
use autodie;
use Bit::Vector;
use Getopt::Long;
use libreriaFinal2;

my $k = 35;
my $mincov = 0;
my $maxcov = 999999;
my %graph;
my %hash;
my $output = '';
my $contig_number = 0;

my $commandline = join ' ', $0, @ARGV;
open my $log_fh, '>>', 'assembler.log';
print {$log_fh} "$commandline\n";
close $log_fh;

if (!GetOptions ('kmer=i'=>\$k, 'mincov=i'=>\$mincov, 'maxcov=i'=>\$maxcov, 'output=s'=>\$output)
or  $output eq '')
    {
       usage();
    }

if ($k%2 == 0) {
  $k++;
}

my $datos;
my $longitud = $k*2 + 20;
my $longitud_seq = $k*2;
my $patron = 'B'. $longitud;

my $total_bytes = int ($longitud / 8) ;
if ($longitud % 8 != 0){
      $total_bytes++;
   }

open my $datosin, '<', 'output_final.bin';
binmode $datosin;

my $counterA = 0;
while(my $nbytes = read $datosin, $datos, $total_bytes){
```

```
   $counterA++;
   if($counterA%10000==0){print $counterA."\n";}

   my $sequencebin = unpack $patron,$datos;
#   print $sequencebin ."\n";
#   my $seq = to_nucleotides(substr $sequencebin, 0, $longitud_seq);
#   my $arrow = oct ("0b" . substr $sequencebin, $longitud_seq, 4);
   my $multi = oct ("0b" . substr $sequencebin, $longitud_seq +4, 16);

   if (($multi <= $maxcov) && ($multi >= $mincov)) {
      my $seq = to_nucleotides(substr $sequencebin, 0, $longitud_seq);
      my $arrow = oct ("0b" . substr $sequencebin, $longitud_seq, 4);
    #  $counterA++;
      $graph{$seq}{$arrow}[0]=$multi;
      $graph{$seq}{$arrow}[1]=0;
#       print $counterA."\t".$seq ."\t" . oct("0b$multi") . "\n";
   }
}

close $datosin;


# Opens new file for graph
open my $graph, '>', "$output.dot";
open my $fasta, '>', "$output.fa";

# Prints header for graph file
print {$graph} "strict digraph $output {\n";
print {$graph} "   node [shape=circle, style=filled];\n";
print {$graph} "   edge [dir=both];\n";
my %assembly;
foreach my $anterior (keys %graph) {
#  print $anterior,"\n";
  foreach my $dato_anterior (keys %{$graph{$anterior}}) {
#      print "\t",$dato_anterior,"\n";
      if ($graph{$anterior}{$dato_anterior}[1]==0){
#                print "\t\t",$graph{$anterior}{$dato_anterior}[1],"\n";
                #$graph{$anterior}{$dato_anterior}[1]=1;
                #my ($a,$b)=alternate_edge($anterior,$dato_anterior, $k);
                #$graph{$a}{$b}[1]=1;
                $contig_number++;
#                print "\t"x3,$anterior,"\t",$dato_anterior,"\t",$k,"\n";
                my ($first, $first_arrow, $last, $last_arrow, $cover, $contig) = assemble
($anterior, $dato_anterior, $k, \%graph);
#                print "hello\n";

$assembly{'>'.$contig_number."\t".$first."\t".$first_arrow."\t".$last_arrow."\t".$last."\t".$cover.
"\t". length($contig)} = $contig ;
                $hash{$first}{$last} += 1;
                #print {$fasta} $contig."\n";
```

```
                my ($arrowtail, $arrowhead);
                if ($first_arrow==0){$arrowtail = 'inv';} else {$arrowtail = 'normal';}
                if ($last_arrow==0){$arrowhead = 'inv';} else {$arrowhead = 'normal';}
                my $color='';
                if (length $contig >= 5000) {$color = "color=\"red\"";}
                if($hash{$first}{$last} > 1 ){
                print {$graph} '   '.$last.' -> '.$first;
                printf    {$graph}    ("    [arrowhead=$arrowtail    arrowtail=$arrowhead    $color
label=%.2f];\n", $cover);
                                print {$graph} '   '.$last." [label=\"\"];\n";
                                print {$graph} '   '.$first." [label=\"\"];\n";
                } else {
                print {$graph} '   '. $first.' -> '.$last;
                printf    {$graph}    ("    [arrowhead=$arrowhead    arrowtail=$arrowtail    $color
label=%.2f];\n", $cover);
                                print {$graph} '   '. $last ." [label=\"\"];\n";
                                print {$graph} '   '. $first ." [label=\"\"];\n";
                }
      }
  }
}

foreach my $contigs (sort {length $assembly{$b} <=> length $assembly{$a}} keys %assembly) {
    print {$fasta} $contigs ."\n";
    print {$fasta} $assembly{$contigs} . "\n";
}

print {$graph} "}\n";
close $graph;
close $fasta;

sub usage
{
print "Unknown option: @_\n" if (@_);
print "\n$0:\n";
print "This program uses a bidirected de Bruijn graph to assemble the reads. Two files are
produced: a fasta \n";
print "file with the contigs and a dot file with the graph.\n\n";
print "Mandatory:\n";
print " --out              STRING  Base name for graph file\n";
print " --kmer             INT     k-mer length\n";
print " --mincov           INT     minimum coverage\n";
print " --maxcov           INT     maximum coverage\n";
exit;
}
```

9.3. Step3.pl

```
#!/usr/bin/env perl
package Assembler;
use strict;
use warnings;
use autodie;
use Bio::SeqIO;
use Getopt::Long;
use libreriaFinal2;

my $k;
my $infile = '';
my %hash;
my %graph;
my $contig_number=0;
my %assembly;

if (!GetOptions ('kmer=i'=>\$k, 'fasta=s' => \$infile) or  $infile eq '') { usage(); }

my $seqio_object = Bio::SeqIO->new(-file => $infile, -format => 'fasta');

while(my $seq_object = $seqio_object->next_seq) {

   my $desc =  $seq_object->desc ;
   if ($desc=~m/^(\S+)\t(\d)\t(\d)\t(\S+)\t\S+\t(\d+)/){
      # select contigs longer than twice the k-mer length)
      if ($5>$k*2) {
         $graph{$1}{"$2$3$4_".$seq_object->id}[0]=$seq_object->seq;
         $graph{$1}{"$2$3$4_".$seq_object->id}[1]=0;
         $graph{$4}{"$3$2$1_".$seq_object->id}[0]=$seq_object->revcom->seq;
         $graph{$4}{"$3$2$1_".$seq_object->id}[1]=0;
      }
   }
}

foreach my $anterior (keys %graph) {
   foreach my $datoanterior (keys %{$graph{$anterior}}) {
      my $destino = substr($datoanterior,2,$k);
      my $llegada = substr($datoanterior,2,1);
      if($graph{$anterior}{$datoanterior}[1]==0){
         $contig_number++;
         my  ($first,  $firstarrow,  $last,  $lastarrow,  $contig)  =  assemble_fasta($anterior,
$datoanterior,$k,\%graph);
      #
$assembly{'>'.$contig_number."\t".$first."\t".$firstarrow."\t".$lastarrow."\t".$last}=$contig;
         print '>'.$contig_number."\t".$first."\t".$firstarrow."\t".$lastarrow."\t".$last . "\n";
         print $contig." \n";
      }
```

```
   }
}

sub usage {
   print "Unknown option: @_\n" if (@_);
   print "\n$0:\n";
   print "This program takes a FastA file produced by Step2.pl and makes a secondary assembly\n\n";
   print "Mandatory:\n";
   print " -f | --fasta    STRING  name of input file in FastA format\n";
   print " -k | --kmer     INT     k-mer length\n";
   exit;
}
```

9.4. libreria.pm

```
package Assembler;
use strict;
use warnings;
use Bit::Vector;
use Bloom::Faster;
use List::Util qw[min max];
use Bio::SeqIO;

#Returns the reverse complement of a nucleotide sequence
sub revcom {
   my $sequence = shift;
   $sequence = reverse $sequence;
   $sequence =~ tr/ATCG/TAGC/;
   return $sequence;
}

#Converts a sequence to a binary value
sub to_binary {
   my $sequence = shift;
   $sequence =~ s/A/00/g;
   $sequence =~ s/T/11/g;
   $sequence =~ s/G/10/g;
   $sequence =~ s/C/01/g;
   return $sequence;
}

sub to_number {
my $number = shift;
my $kmer_length = shift;
my $valor = Bit::Vector->new_Dec ($kmer_length*2, $number);
$number = $valor->to_Bin();
return $number;
}

#Converts a bit vector to a nucleotide sequence
sub to_nucleotides {
   my $sequence = '';
   my $number = shift;
   while ($number =~ s/^(\S\S)//) {
             if ($1 eq '00') {
             $sequence = $sequence.'A';
             }elsif ($1 eq '01') {
             $sequence = $sequence. 'C';
             } elsif ($1 eq '10') {
             $sequence = $sequence.'G';
             } else {
             $sequence = $sequence.'T';
```

```
            }
  }
  return $sequence;
}

#Returns the direction of arrow at the beginning of an edge
sub beginning {
   my $number = shift;
   if ($number & 8) {
     return 1;
   } else {
     return 0;
   }
}

#Returns the direction of arrow at the end of an edge
sub end {
   my $number = shift;
   if ($number & 4) {
      return 1;
   } else {
      return 0;
   }
}

#Determines the base to add while moving from a node to the next
sub next_base {
   my $number = shift;
   my $base = $number & 3;
   if ($base == 0) {
        return 'A';
   } elsif ($base == 1) {
        return 'C';
   } elsif ($base == 2) {
        return 'G';
   } else {
        return 'T';
   }
}

sub assemble {
   my ($source_node, $source_edge_info, $kmer_length, $graph_ref) = @_;
   my ($first, $last, $first_arrow, $last_arrow);
   my $multip = 0;
   my $nedges = 0;
   my $assembly_2;
   my $current_node = $source_node;
   my $current_edge_info = $source_edge_info;
   my $current_node_sequence = $current_node;
   my $beginning = beginning ($current_edge_info);
```

```
if ($beginning == 1) { $current_node_sequence  = revcom ($current_node_sequence); }

my $assembly = $current_node_sequence;
while ( defined $graph_ref->{$current_node}{$current_edge_info}){
   my $current_node_sequence = $current_node;
   my $next_base = next_base ($current_edge_info);
   $assembly = $assembly . $next_base;
   $graph_ref->{$current_node}{$current_edge_info}[1]=1;
   my ($a,$b)=alternate_edge($current_node,$current_edge_info, $kmer_length);
   $graph_ref->{$a}{$b}[1]=1;
   $nedges++;
   $multip = $multip + $graph_ref->{$current_node}{$current_edge_info}[0];
   my $end = end ($current_edge_info);
   my $beginning = beginning ($current_edge_info);
   if ($beginning == 1) { $current_node_sequence  = revcom ($current_node_sequence);}
   my $sink_node_sequence = substr ($current_node_sequence, 1,$kmer_length-1) . $next_base;
   my $revcom_sink = revcom ($sink_node_sequence);
   if ($sink_node_sequence gt $revcom_sink) {$sink_node_sequence = $revcom_sink;}
   my $sink_node = $sink_node_sequence;
   if (defined $graph_ref->{$sink_node}){
      my $counter=0;
      my $counter2=0;
      my $sink_edge_info;
      foreach my $value (keys %{$graph_ref->{$sink_node}}){
         my $beginning = beginning($value);
         if ($beginning != $end){
            $counter++;
            $sink_edge_info = $value;
         }
         my $newedge;
         if ($beginning == 1){
            $newedge  = revcom($sink_node_sequence) . next_base($value);
         } else {
            $newedge  = $sink_node_sequence . next_base($value);
         }
         if(substr($newedge,0,$kmer_length) eq revcom(substr $newedge,1,$kmer_length)){
            $counter2=$counter2+2;
         } else{
            $counter2++;
         }
      }
      if ($counter ==1 && $counter2 == 2 && $graph_ref->{$sink_node}{$sink_edge_info}[1]==0){
         $current_node = $sink_node;
         $current_edge_info = $sink_edge_info;
      } else{
         $first = $sink_node;
         $first_arrow = end($current_edge_info);
         last;
      }
   }
```

```
   }
   ($current_node, $current_edge_info) = alternate_edge($source_node, $source_edge_info,
$kmer_length);
   while (defined $graph_ref->{$current_node}{$current_edge_info}){
      my $current_node_sequence = $current_node;
      my $next_base = next_base ($current_edge_info);
      $assembly_2 = $assembly_2 . $next_base;
      $graph_ref->{$current_node}{$current_edge_info}[1]=1;
      my ($a,$b)=alternate_edge($current_node,$current_edge_info, $kmer_length);
      $graph_ref->{$a}{$b}[1]=1;
      $nedges++;
      $multip = $multip + $graph_ref->{$current_node}{$current_edge_info}[0];
      my $end = end ($current_edge_info);
      my $beginning = beginning ($current_edge_info);
      if ($beginning == 1) {$current_node_sequence  = revcom ($current_node_sequence);}
      my $sink_node_sequence = substr ($current_node_sequence, 1,$kmer_length-1) . $next_base;
      my $revcom_sink = revcom ($sink_node_sequence);
      if ($sink_node_sequence gt $revcom_sink) {
         $sink_node_sequence = $revcom_sink;
      }
      my $sink_node = $sink_node_sequence;
      if (defined $graph_ref->{$sink_node}){
         my $counter =0;
         my $counter2 =0;
         my $sink_edge_info;
         foreach my $value (keys %{$graph_ref->{$sink_node}}){
            my $beginning = beginning($value);
            if ($beginning != $end){
               $counter++;
               $sink_edge_info = $value;
            }
            my $newedge;
            if ($beginning == 1){
               $newedge  = revcom($sink_node_sequence) . next_base($value);
            } else {
               $newedge  = $sink_node_sequence . next_base($value);
            }
            if(substr $newedge,0,$kmer_length eq revcom(substr $newedge,1,$kmer_length)){
               $counter2=$counter2+2;
            } else {
               $counter2++;
            }
         }
         if ($counter == 1 && $counter2 == 2 && $graph_ref->{$sink_node}{$sink_edge_info}[1]==0){
            $current_node = $sink_node;
            $current_edge_info = $sink_edge_info;
         } else {
            $last=$sink_node;
            $last_arrow = end($current_edge_info);
            last;
```

```
         }
      }
   }
   return $first, $first_arrow, $last, $last_arrow, $multip/$nedges,
revcom($assembly).substr($assembly_2, 1, length($assembly_2) - 1);
}

sub alternate_edge {
   my ($current_node, $current_edge_info, $kmer_length) = @_;
   my $current_node_sequence = $current_node;
   my $beginning = beginning ($current_edge_info);
   if ($beginning == 1) {$current_node_sequence  = revcom ($current_node_sequence);}
   my $otherway_node_sequence = substr ($current_node_sequence, 1,($kmer_length-1)) . next_base
($current_edge_info);
   my $otherway_edge_info;
   my $revcom_otherway = revcom ($otherway_node_sequence);
   if ($otherway_node_sequence gt $revcom_otherway) {$otherway_node_sequence = $revcom_otherway;}
   $beginning = end($current_edge_info);
   my $end = beginning($current_edge_info);
   if ($end == 0) {
      $otherway_edge_info = $beginning.'0'. to_binary(revcom(substr $current_node, 0 , 1));
   } else {
      $otherway_edge_info = $beginning.'1'. to_binary(substr $current_node, $kmer_length-1,1);
   }
   return $otherway_node_sequence, oct "0b$otherway_edge_info";
}

sub dsk {
   my $filein = shift;
   my $formatin = shift;
   my $kvalue = shift;
   open my $filehandle, "zcat $filein | " or die $!;
   open my $resultado, ">resultado_dsk.txt";
   my $niters = 10;
   my $nparts = 10;
   for (my $iter = 0; $iter<=$niters-1; $iter++) {
      open my $filehandle, "zcat $filein | " or die $!;
      my $seqio = Bio::SeqIO->new(-fh => $filehandle, -format => $formatin);
      print "Iteration $iter\n";
      my @fh;
      for (my $part = 0; $part<=$nparts-1; $part++) { open $fh[$part], ">archivo$part.txt"; }
      while (my $seqobject = $seqio->next_seq) {
         my $seq = $seqobject->seq;
         if ($seq=~m/N/g){ next; }
         for (my $pos = 0; $pos <= length($seq) - $kvalue; $pos++){
            my $kmer = substr($seq,$pos,$kvalue);
            if ($kmer ge revcom($kmer)){ $kmer = revcom($kmer);}
            if(hash($kmer) % $niters == $iter){
               my $j = int(hash($kmer) / $niters) % $nparts;
               print {$fh[$j]} $kmer."\n";
```

```
            }
         }
      }
      for (my $j=0; $j<=$nparts-1; $j++) { close $fh[$j]; }
      for (my $part = 0; $part <= $nparts-1; $part++) {
         print "   ..counting in partition $part\n";
         open my $fileopener, "<archivo$part.txt";
         my %T;
         while(my $linea = <$fileopener>){
            chomp $linea;
            if(defined $T{$linea}){
              $T{$linea}++;
            } else {
              $T{$linea}=1;
            }
         }
         foreach my $kmer (keys %T){
            print {$resultado} $kmer . "\t" . $T{$kmer} . "\n";
         }
         close $fileopener;
      }
   }
   close $resultado;
   open my $fileres, "<resultado_dsk.txt";
   my %all;
   while (my $line = $fileres ) {
      if($line =~m/^(\S+)\t(\D+)/) { $all{$1} = $2; }
   }
   return \%all;
}

sub hash {
   my $kmer2 = shift;
   my $k = length $kmer2;
   $kmer2 =~s/A/00/g;
   $kmer2 =~s/C/01/g;
   $kmer2 =~s/G/10/g;
   $kmer2 =~s/T/11/g;
   my $length = min(32,$k*2);
   $kmer2 = substr($kmer2,0,$length);
   my $number = oct "0b$kmer2";
   return $number;
}

sub turtle {  #nueva subrutina
   my $filein = shift;
   my $formatin = shift;
   my $kvalue = shift;
   open my $filehandle, "zcat $filein | " or die $!;
   my $seqio_object = Bio::SeqIO->new(-format => $formatin, -fh => $filehandle);
```

```
my $filtro = new Bloom::Faster({n => 7000000000000, e => 0.0001});
my $contador = 0;
my @kmeros;
while (my $lectura = $seqio_object->next_seq) {
   $contador++;
   my $secuencia = $lectura->seq;
   for my $i (0 .. length($secuencia) - $kvalue){
      my $kmer = substr($secuencia, $i, $kvalue);
      if ($kmer ge revcom($kmer)){ $kmer = revcom($kmer);}
      if ($filtro->add($kmer)) {
         push @kmeros, "$kmer-1";
      }
   }

   if ($contador%100000==0) {
      print $contador."\n";
      my @sorted = sort {$a cmp $b} @kmeros;
      my @nuevo;
      my $conteo = 0;
      my $previo = '';
      my $contadorgeneral;
      foreach my $element (@sorted) {
          $element=~m/(\w+)\-*(\d*)/;
          $contadorgeneral++;
          if ($1 eq $previo) {
             $conteo = $conteo + $2;
          } else {
             if($contadorgeneral !=1) {push @nuevo, $previo.'-'.$conteo;}
             $conteo = $2;
          }
         $previo = $1;
      }
      push @nuevo, $previo.'-'.$conteo ;
      @kmeros = @nuevo;
   }
}
my @sorted = sort {$a cmp $b} @kmeros;
my @nuevo;
my $conteo = 0;
my $previo = '';
my $contadorgeneral;
foreach my $element (@sorted) {
   $element=~m/(\w+)\-*(\d*)/;
   $contadorgeneral++;
   if ($1 eq $previo) {
      $conteo = $conteo + $2;
   } else {
      if($contadorgeneral !=1) {push @nuevo, $previo.'-'.$conteo;}
      $conteo = $2;
   }
```

```
         $previo = $1;
      }
   push @nuevo, $previo.'-'.$conteo ;
   my %T;
   foreach my $element (@nuevo) {
      $element=~m/(\w+)\-*(\d*)/;
      $T{$1}= $2 + 1; #quitar el +1 ?
   }
   return \%T;
}

sub bfcounter {
   my $filein = shift;
   my $format_in = shift;
   my $k_value = shift;
   open my $inFh, "zcat $filein | head -n 10000|" or die $!;
   my $seqio_object = Bio::SeqIO->new(-fh => $inFh, -format => $format_in);
   my %T;
   my $bloom = new Bloom::Faster({n => 7000000000000, e => 0.0001});
   while (my $seq_object = $seqio_object->next_seq) {
       my $read = $seq_object->seq;
       next if $read =~m/N/;
       my $lonread = (length $read) -$k_value;
       for my $pos (0 .. $lonread) {
          my $edge = substr $read, $pos, $k_value;
          if ($edge ge revcom($edge)){ $edge = revcom($edge);}
          if($bloom->add($edge)){ $T{$edge}=0; }
       }
   }
   close $inFh;
   open my $inFh2, "zcat $filein | head -n 10000|" or die $!;
   my $seqio_object2 = Bio::SeqIO->new(-fh => $inFh2, -format => $format_in);
   while (my $seq_object = $seqio_object2->next_seq) {
      my $read = $seq_object->seq;
      next if $read =~m/N/;
      my $lonread = (length $read) -$k_value;
      for my $pos (0 .. $lonread) {
         my $edge = substr $read, $pos, $k_value;
         if ($edge ge revcom($edge)){ $edge = revcom($edge);}
         if(defined $T{$edge}){ $T{$edge}= $T{$edge} + 1; }
      }
   }
   close $inFh2;
   foreach my $keys (keys %T) {
      if ($T{$keys} < 1) { delete $T{$keys}; }
   }
   return \%T;
}

sub turtleFast {
```

```
   my $filein = shift;
   my $formatin = shift;
   my $kvalue = shift;
   my $period;
   my $resto;
   my $niters = 10;
   open my $resultados, ">resultados.txt";
   for(my $iter =0; $iter<=$niters-1; $iter++) {
      open my $filehandle, "zcat $filein | " or die $!;
      if ($formatin eq 'fastq'){$period = 4; $resto = 2;} elsif($formatin eq 'fasta'){$period = 2;
$resto=0;}
      my $filtro = new Bloom::Faster({n => 7000000000000, e => 0.0001});
      my $contador = 0;
      my @kmeros;
      my $nreads = 0;
      while (my $lectura = <$filehandle>) {
         $contador++;
         if($contador%$period == $resto) {
            chomp $lectura;
            next if $lectura=~m/N/;
            $nreads++;
            my $secuencia = $lectura;
            for my $i (0 .. length($secuencia) - $kvalue){
               my $kmer = substr($secuencia, $i, $kvalue);
               if(hash($kmer)%$niters == $iter){
                  if ($kmer ge revcom($kmer)){ $kmer = revcom($kmer);}
                  if ($filtro->add($kmer)) {
                     push @kmeros, "$kmer-1";
                  }
               }
            }
            if ($nreads%200000 == 0) {
               print $nreads."\n";
               my @sorted = sort {$a cmp $b} @kmeros;
               my @nuevo;
               my $conteo = 0;
               my $previo = '';
               my $contadorgeneral;
               foreach my $element (@sorted) {
                  $element=~m/(\w+)\-*(\d*)/;
                  $contadorgeneral++;
                  if ($1 eq $previo) {
                     $conteo = $conteo + $2;
                  } else {
                     if($contadorgeneral !=1) {push @nuevo, $previo.'-'.$conteo;}
                     $conteo = $2;
                  }
                  $previo = $1;
               }
               push @nuevo, $previo.'-'.$conteo ;
```

```
                @kmeros = @nuevo;
            }
         }
      }
      my @sorted = sort {$a cmp $b} @kmeros;
      my @nuevo;
      my $conteo = 0;
      my $previo = '';
      my $contadorgeneral;
      foreach my $element (@sorted) {
         $element=~m/(\w+)\-*(\d*)/;
         $contadorgeneral++;
         if ($1 eq $previo) {
            $conteo = $conteo + $2;
         } else {
            if($contadorgeneral !=1) {push @nuevo, $previo.'-'.$conteo;}
            $conteo = $2;
         }
         $previo = $1;
      }
      push @nuevo, $previo.'-'.$conteo ;
      my %T;
      foreach my $element (@nuevo) {
         $element=~m/(\w+)\-*(\d*)/;
         $T{$1}=$2+1; #quitar el +1 ?
      }
      foreach my $clave (keys %T){
         print {$resultados} $clave."\t". $T{$clave} . "\n";
      }
      close $filehandle;
   }
   close $resultados;
   open my $again, "<resultados.txt";
   my %finalcount;
   while(my $linea=<$again>){
      if($linea=~m/(\S+)\t(\d+)/){ $finalcount{$1} = $2; }
   }
   close $again;
   return \%finalcount;
}

sub bfcounter2 {
   my $filein = shift;
   my $formatin = shift;
   my $k_value = shift;
   my $period;
   my $resto;
   my $niters = 5;
   my $contador = 0;
   open my $resultados, ">resultados.txt";
```

```
   if ($formatin eq 'fastq'){$period = 4; $resto = 2;} elsif($formatin eq 'fasta'){$period = 2;
$resto=0;}
   for(my $iter =0; $iter<=$niters-1; $iter++) {
      open my $filehandle, "zcat $filein |  " or die $!;
      $contador = 0;
      print "Iteracion $iter\n";
      my %T;
      my $bloom = new Bloom::Faster({n => 7000000000000, e => 0.0001});
      print "Step 1\n";
      while (my $read = <$filehandle>) {
         $contador++;
         if($contador%1000000==0){print "$contador\n";}
         if($contador%$period == $resto) {
            next if $read =~m/N/;
            chomp $read;
            my $lonread = (length $read) -$k_value;
            for my $pos (0 .. $lonread) {
               my $edge = substr $read, $pos, $k_value;
               if ($edge ge revcom($edge)){ $edge = revcom($edge);}
               if(hash($edge)%$niters == $iter){
                  if($bloom->add($edge)){ $T{$edge}=0; }
               }
            }
         }
      }
      close $filehandle;
      print "Step 2\n";
      open my $filehandle2, "zcat $filein | " or die $!;
      $contador = 0;
      while (my $read = <$filehandle2>) {
         $contador++;
         if ($contador%1000000 == 0) {print $contador . "\n";}
         if($contador%$period == $resto) {
            next if $read =~m/N/;
            my $lonread = (length $read) - $k_value;
            for my $pos (0 .. $lonread) {
               my $edge = substr $read, $pos, $k_value;
               if ($edge ge revcom($edge)){ $edge = revcom($edge);}
               if(defined $T{$edge}){ $T{$edge}= $T{$edge} + 1; }
            }
         }
      }
      close $filehandle;
      foreach my $keys (keys %T) {
         if ($T{$keys} <= 1) {   #corregido a menor o igual (ver que dice paper)
            delete $T{$keys};
         } else {
            print {$resultados} $keys."\t". $T{$keys} . "\n";
         }
      }
```

```
   }#close iters
   close $resultados;
   open my $again, "<resultados.txt";
   my %finalcount;
   while(my $linea=<$again>){
      if($linea=~m/(\S+)\t(\d+)/) {
         $finalcount{$1}=$2;
      }
   }
   close $again;
   return \%finalcount;
}

sub assemble_fasta {
   my ($source_node, $source_edge_info, $k, $graphref) = @_;
   my $waytogo;
   my ($first, $last, $firstarrow, $lastarrow);
   my $current_node = $source_node;
   my $current_edge_info = $source_edge_info;
   my $current_node_sequence = $current_node;
   my $assembly = $graphref->{$current_node}{$current_edge_info}[0];
   my $llegada='';
   while( defined $graphref->{$current_node}{$current_edge_info}){
      $graphref->{$current_node}{$current_edge_info}[1]=1;
      $graphref-
>{substr($current_edge_info,2,$k)}{substr($current_edge_info,1,1).substr($current_edge_info,0,1).$c
urrent_node.substr($current_edge_info,$k+2,length($current_edge_info)-($k+2))}[1]=1;
      my $llegada = substr($current_edge_info,1,1);
      my $verticellegada = substr($current_edge_info,2,$k);
      my $nsalidas=0;
      my $ncaminos=0;
      foreach my $salidas (keys %{$graphref->{$verticellegada}}){
         $nsalidas++;
         if($llegada != substr($salidas,0,1)){
            $ncaminos++;
            $waytogo = $salidas;
        }
      }
      if(defined($waytogo) && $nsalidas==2 && $ncaminos== 1 && $graphref-
>{$verticellegada}{$waytogo}[1] == 0 ){
               if ($llegada==1) {
                    $assembly = $assembly . substr($graphref-
>{$verticellegada}{$waytogo}[0],$k,length($graphref->{$verticellegada}{$waytogo})-$k);
               } else {
                    $assembly = $assembly . substr(revcom($graphref-
>{$verticellegada}{$waytogo}[0]),$k,length($graphref->{$verticellegada}{$waytogo})-$k);
              }
              $current_node = $verticellegada;
              $current_edge_info = $waytogo;
      } else {
```

```
           $first = $verticellegada;
           if (defined($waytogo)){ $firstarrow = substr($waytogo,1,1);} else
{$firstarrow=$llegada;}
           last;
      }
   }

   my $assembly2='';
   $current_node=substr($source_edge_info,2,$k);

$current_edge_info=substr($source_edge_info,1,1).substr($source_edge_info,0,1).$source_node.substr(
$source_edge_info,$k+2,length($source_edge_info)-($k+2));
   while( defined $graphref->{$current_node}{$current_edge_info}){
      $graphref->{$current_node}{$current_edge_info}[1]=1;
      $graphref-
>{substr($current_edge_info,2,$k)}{substr($current_edge_info,1,1).substr($current_edge_info,0,1).$c
urrent_node.substr($current_edge_info,$k+2,length($current_edge_info)-($k+2))}[1]=1;
      my $llegada = substr($current_edge_info,1,1);
      my $verticellegada = substr($current_edge_info,2,$k);
      my $nsalidas=0;
      my $ncaminos=0;
      foreach my $salidas (keys %{$graphref->{$verticellegada}}){
         $nsalidas++;
         if($llegada != substr($salidas,0,1)){
            $ncaminos++;
            $waytogo = $salidas;
        }
      }
      if(defined($waytogo) && $nsalidas==2 && $ncaminos== 1 && $graphref-
>{$verticellegada}{$waytogo}[1] == 0 ){
               if ($llegada==1) {
                    $assembly2 = $assembly2 . substr($graphref-
>{$verticellegada}{$waytogo}[0],$k,length($graphref->{$verticellegada}{$waytogo})-$k);
               } else {
                    $assembly2 = $assembly2 . substr(revcom($graphref-
>{$verticellegada}{$waytogo}[0]),$k,length($graphref->{$verticellegada}{$waytogo})-$k);
              }
              $current_node = $verticellegada;
              $current_edge_info = $waytogo;
      } else {
           $last = $verticellegada;
           if (defined($waytogo)){ $lastarrow = substr($waytogo,1,1);} else {$lastarrow=$llegada;}
           last;
      }
   }
   return $first,$firstarrow,$last,$lastarrow,revcom($assembly).$assembly2;
}

1;
```